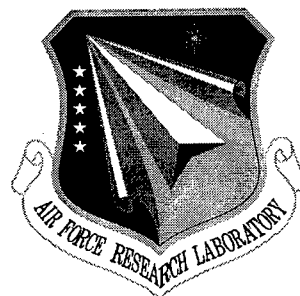AFRL-IF-RS-TR-1999-181
Final Technical Report
August 1999

A MULTIMODELING FRAMEWORK FOR DISTRIBUTED SIMULATION

University of Florida

Paul A. Fishwick, Robert M. Cubert, and Kangsun Lee

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

19991022 012

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.
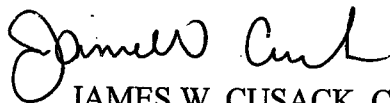
AFRL-IF-RS-TR-1999-181 has been reviewed and is approved for publication.

APPROVED:

ALEX F. SISTI
Project Engineer

FOR THE DIRECTOR:

JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED	
	August 1999	Final	Sep 95 - Aug 97

4. TITLE AND SUBTITLE
A MULTIMODELING FRAMEWORK FOR DISTRIBUTED SIMULATION

5. FUNDING NUMBERS
C: F30602-95-C-0267
PE: 62702F
PR: 4594
TA: 15
WU: N5

6. AUTHOR(S)
Paul A Fishwick, Robert M Cubert, Kangsun Lee

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Florida
E301 Computer Sciences & Engineering Building
P. O. Box 116120
Gainesville FL 32611-6120

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFSB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-1999-181

11. SUPPLEMENTARY NOTES
AFRL/IF Project Engineer: Alex F. Sisti/(315) 330-3983

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report represents a compendium of previously published reports generated during the conduct of this contract. They describe different aspects of an application framework for modeling and simulation; called Object-Oriented Physical Multimodeling (OOPM). It extends object-oriented program design with visualization and a definition of systems modeling that reinforces the relationship of model to program.

14. SUBJECT TERMS
Multimodeling, Model Repository, Web-based Simulation, Model Abstraction, Object-Oriented Simulation, Physical Modeling

15. NUMBER OF PAGES
92

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

Acknowledgements

Paul A. Fishwick
Robert M. Cubert
Kangsun Lee

Introduction

This report represents a compendium of previously published reports generated during the conduct of Air Force contract F30602-95-C-0267. They describe different aspects of an application framework for modeling and simulation; called Object-Oriented Physical Multimodeling (OOPM), it extends object-oriented program design with visualization and a definition of systems modeling that reinforces the relationship of model to program.

Alex F. Sisti

A FRAMEWORK FOR DISTRIBUTED OBJECT-ORIENTED MULTIMODELING AND SIMULATION

Robert M. Cubert
Paul A. Fishwick

Department of Computer and Information Science and Engineering
University of Florida
CSE Building, Room E301
Gainesville FL 32611-6120, U.S.A.

ABSTRACT

We have developed a multimodeling object-oriented (OO) simulation environment (MOOSE), which is a framework for modeling and developing simulation software. Its architecture derives from Object Oriented Physical Modeling (OOPM), which extends classical object-oriented methodology to allow attributes and methods to take on *models* as values. The MOOSE Model Repository (MMR) allows distributed model definitions, and so supports "web-based simulation", integrated with the web and made available on the Internet. MOOSE features multimodeling, an OO approach to model refinement and abstraction, allowing creation of heterogeneous hierarchical models. Dynamic models comprising multimodels include Finite State Machines, Functional Block Models, Equation Constraint Models, and Rule Based Models. MOOSE emphasizes visualization, & effective use of OO metaphors to connect conceptual model to program, and to capture model geometry and dynamics. The MOOSE human-computer interface has two GUI's: *Modeler*, for model design, and *Scenario*, for model execution control and visualization. MOOSE back end generates a model description in a target language such as C++, then translates and adds runtime support to form an Engine. Model execution consists of Engine running synchronously with Scenario. The MOOSE approach facilitates model development, models with greater intuitive appeal, communication among model authors, better agreement between simulation programs and their conceptual models, component reuse, and model/program extensibility.

1 INTRODUCTION

The World Wide Web (often just referred to as "the web") represents a fertile area for computer simulation research. Combining the web with computer simulation can have a key impact on future simulation research. Among the directions one can take in this endeavor are (1) parallel and distributed model execution, and (2) distributed model repositories. Both these avenues are fruitful. We have narrowed our focus to the area of distributed model repositories since there has been less research in this area than in the more mature field of distributed simulation (Fujimoto, 1990; Lin and Fishwick, 1996). Also, the concept of model repository lends itself to the study of how to organize model information. Since the web is also concerned with how to effectively organize information, this appears to be a reasonable way to blend the web with simulation. The web defines a networked hypermedia approach to storing information. Search engines exist to help a user browse or perform a topical search. In simulation, information is generally focused on physical objects. These physical objects, whether they are humans, milling machines or a container of fluid, have attributes and exhibit behaviors. If we are to permit a situation where physical object information is as freely available as hypermedia to remote users on today's web, then we need to (1) formalize this information, (2) provide a way to integrate to today's web-based information, and (3) effect mechanisms for searching and browsing models. In this paper we explore these three issues in the context of OOPM and MOOSE.

MOOSE is an acronym for "Multimodel Object Oriented Simulation Environment", a modeling and simulation framework under development at University of Florida. MOOSE is an implementation of "Object Oriented Physical Modeling" (OOPM) (Fishwick 1996), which is an approach to modeling and simulation which defines a formal approach to capturing physical knowledge in a form that extends the object design principles specified in the fast-growing area of object design within software engineering and programming language design (Booch, 1994; Rumbaugh et al., 1991) Some of the current object-oriented design methodology requires modification to support physical modeling. Moreover, there does not cur-

rently exist a clearly-defined method of capturing physical knowledge in an object-oriented modeling framework even though many of the object-oriented "nuts and bolts" exist to help structure the method. The OOPM methodology satisfies the requirement of development of a theoretical framework for physical modeling, while allowing for legacy code insertion and user-defined dynamic model and multimodel types.

Initial development of MOOSE, focussed on an environment consisting of a single host system, has been completed, with results reported in detail by Cubert and Fishwick (1997a). The next step, now underway, involves expanding the environment to permit model definitions to be distributed over any number of hosts within the framework of the worldwide web. There are two kinds of distributed operation to consider: one is where model definitions are distributed, with some classes defined here, others there; the second is where model execution proceeds as a distributed simulation, executing simultaneously on a number of hosts, with one object instantiated here, another there. The MOOSE architecture supports both kinds of distributed operation; with our emphasis being on distributing definition of multimodels.

We first briefly summarize some of MOOSE's focal ideas and properties, such as use of multimodels to facilitate model refinement to achieve appropriate levels of model fidelity, use of dynamic models, and reuse by design. Fuller treatment of these topics, as well as issues such as how MOOSE captures the geometry of a model, relation between conceptual model and simulation program, relations such as aggregation, containment, composition, usage, association, generalization, and specialization, validation and verification, extensibility, speed of development, and platforms and portability, have been addressed by the authors elsewhere (Cubert and Fishwick, 1997b). After presenting background on the components of MOOSE and how they interact, in sufficient detail to orient the reader, the major emphasis will focus on MOOSE Model Repository (MMR), because this is the vehicle which expands the horizons of MOOSE to the limits of the web.

Thus the balance of the paper is organized as follows. In Section 2 we briefly present focal issues such as multimodels, dynamic models, and reuse. Section 3 covers the components of MOOSE and how they interact. Section 4 goes into detail on MMR and distributed operation. Section 5 presents our conclusions and directions for future work.

2 MULTIMODELS, DYNAMIC MODELS, AND REUSE BY DESIGN

Derived from OOPM principles, MOOSE promises not only to tightly couple a model's human author into the modeling and simulation process through an intuitive human–computer interface (HCI), but also to help a model author to perform any or all of the following: (1) to think clearly about, to better understand, or to elucidate a model; (2) to participate in a collaborative modeling effort; (3) to repeatedly and painlessly refine a model as required, in order to achieve adequate fidelity at minimal development cost; (4) to painlessly build large models out of existing working smaller ones; (5) to start with a conceptual model which is intuitively clear to domain experts, and to unambiguously and automatically convert this to a simulation program; (6) to create or change a simulation program without being a programmer; (7) to perform simulation model execution and to present simulation results in a meaningful way so as to facilitate the other objectives above.

The degree of detail in a model reflects the model author's abstraction perspective (Fishwick, 1988). Refinement to greater detail is used to obtain model fidelity that is adequate in the eyes of the model author from a given abstraction perspective (Fishwick 1989), and with certain objectives for the model or simulation to meet (Berzins 1986). MOOSE addresses this area with *multimodeling*, an approach which glues together models of the same or different types, produced during the activity of model refinement, and reflecting various abstraction perspectives (Fishwick and Lee, 1996). Refinement can be adjustable during model execution as well as during model design. The pieces that are put together to form a model, such as described above, are *dynamic models*. Dynamic model types supported include Finite State Machine (FSM), Functional Block Model (FBM), Equation Constraint Model (EQN), and Rule-based Model (RBM); alternatively, users may create their own C++ "code models"; model types may be freely combined. The dynamic model types implemented so far form a popular collection of approaches used in simulation (Fishwick, 1995); additional dynamic model types will likely be added to the MOOSE repertoire; MOOSE has been designed to be extensible in this regard. In addition to model refinement during development, multimodeling may also be used during model execution: components of a multimodel may be behaviorally abstracted to fit time constraints placed upon model execution.

In MOOSE, dynamic behavior of the system is represented by *dynamic models*. Dynamic models are methods of the various classes in the conceptual

model. Dynamic models are readily added, changed, and removed, as part of model development, at any time. Here MOOSE makes good its promise to the model author to be able to create or change a simulation program without being a programmer. MOOSE presently incorporates several kinds of dynamic model: FBM, FSM, EQN, and RBM, with others contemplated, such as Petri nets, and System Dynamics models. From this ensemble of popular and capable dynamic model types, the model author picks one or more dynamic model types to define methods of the classes of the model. Construction of each specific dynamic model typically involves drawing the kinds of "pictures" that people tend to make on the back of an envelope or a blackboard when informally describing a model to someone else. The MOOSE HCI facilitates these constructions: allowing the model author to specify components, connect components, provide inputs, outputs, conditions, and so forth.

To support the kind of heterogeneous model hierarchies shown abstractly in Figure 1, we must ensure that our models are *closed under coupling*. In short, this suggests that the method of coupling one model component to another must be clearly defined. Two kinds of coupling exist: intralevel and interlevel. Intralevel coupling reflects model components coupled to one another in the same model. For example, one needs to specify rules of how Petri nets, compartmental models and System Dynamics graphs are formed. With a System Dynamics graph, a rule of model building defines that any level has an input rate and an output rate. A more interesting case arises in interlevel coupling since we must ensure that we define rules as to how model components from one model can be refined into models of different types. Can a finite state machine state be refined into a Petri net, or can a functional block model contain finite state machines (FSM) inside blocks? What are the rules to guide this refinement? The rule for intralevel coupling is based on functional composition. The primitive of *function* with its *input* and *output* defines the coupling procedure in the following way. All models are encapsulated in a single function. This represents the outer shell to support interlevel coupling. Within a model there are *functional entry points*. These are inner shells where new models may be optionally inserted. Each model type has its own entry point defined differently. For example, for the model type "FSM", we may define each state to be of the form: $v(state) = f()$ where $f()$ is an arbitrary function and $v(state)$ defines the value of the attribute *state*. If *state* is not refined, then $f()$ returns the value of the state as a character string or integer. If *state* is refined, then $f()$ may be replaced by *any* function—whether this function is a dynamic
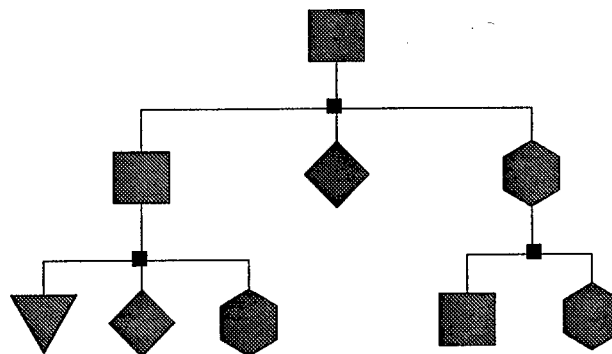

Figure 1: Multimodeling Tree Structure for Model Refinement; Polygons Depict the Heterogeneous Nature of Multimodeling: each type of Polygon represents one type of Dynamic Model

model or a code method. The coupling approaches are defined in more detail by Fishwick (1997).

Reuse of one's own previous work, as well as by one model author of the work of others, is encouraged by availability of model repositories. An application framework such as MOOSE is more than just a class library. In an application framework, classes from the library are related in such a way that a class is not used in isolation but within a design encouraged and supported by the framework. The MOOSE Model Repository (MMR) is aptly named because it is not just a class library; as a model repository, it stores not only a collection of classes available for reuse, but also the design which relates those classes as to how they play together within the geometry and dynamics of a particular model. This enables support for one of Booch's (1994) five attributes of a complex system: "A complex system that works is invariably found to have evolved from a simpler system that worked .... A complex system designed from scratch never works and cannot be patched up to make it work.". Using MMR, model authors can start from some piece of their overall system that happens to appeal to them intuitively. When several such pieces are working, they may be combined into a more-complex (working) system.

3 COMPONENTS OF MOOSE

Components of MOOSE fall into three groups: Human Computer Interface (HCI), Library, and Back End. The HCI is comprised of two components: Modeler and Scenario. *Modeler* interacts with the human model author via graphical user interface (GUI) to construct the model. In simulation parlance, this is *model design*. Modeler relies on the Library (discussed below) to store model definitions. *Scenario* is
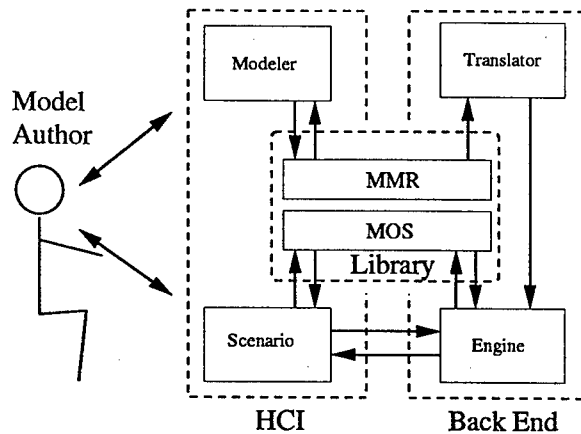
Figure 2: The three Components of MOOSE (HCI, Library, and Back End) shown outlined with dashed line Boxes; Parts within each Component are shown outlined with solid Boxes

a visualizer employing a GUI. Scenario activates and initializes simulation model execution (which we call Engine) at the behest of user (who may or may not be the original model author). Scenario maintains synchronous interaction with Engine, visualizing Engine output in a form meaningful to user, optionally allowing user to interact with Engine, including modifying simulation parameters and changing the rate of simulation progress.

Modeler GUI's "main" part defines classes and objects and relations among classes (aggregation and specialization or generalization) on one or more canvases. On the canvas, rectangles represent classes. These rectangles are joined by relations to form a tree, or, more generally, a graph, reflecting relations in the system being modeled. Some models look cleaner if aggregations and specializations are kept on separate canvases; this is supported but not required. Similarly, some models are large enough that several canvases are needed to capture the representation. Each class is a box which, when opened, reveals more information, and permits the model author to define the name of the class, its attributes, its methods, and its named objects. Within each method, the model author may specify input parameters and output parameters, as well as identifying which dynamic model type the method is to be. In addition to the "main" GUI presented above, there is a GUI editor for each dynamic model type, *i.e.*: the FSM editor for finite state machines, the FBM editor for functional block models, the EQN editor for equation constraint models, and the RBM editor for rule-based models.

The Back End has two components: Translator and Engine. *Translator* is a bridge between model design and model execution: Translator reads from

the Library a language-neutral model definition produced by Modeler, and emits a complete computer program for the model, in Translator Target Language (TTL). Presently MOOSE TTL is C++; potentially, TTL can be Java or another language. This simulation program emitted by Translator in TTL is called *Engine*. Once compiled and linked with runtime support, the Engine executable is activated under control of Scenario to perform model execution. Library has two components: *MOOSE Model Repository (MMR)* and *MOOSE Object Store (MOS)*. MOS holds object data and MMR holds object metadata. MMR keeps track of models as they are being built. MMR servers provide a database management system (DBMS) for model definitions. MMR clients work with Modeler and Translator to define and use model definitions. Models and model components created by other model authors (or the same model author previously) are available for browsing, inclusion, and/or reuse. Base classes such as sets for modeling collections and popular geometries for spatial models are available to the model author. An MMR client can simultaneously maintain conversations with several MMR servers on different hosts, thus permitting model definitions to be distributed. An MMR Server can simultaneously maintain conversations with several MMR clients, on the same or different hosts, which permits collaboration on model development. MOS does for objects much of what MMR does for models. MOS works with Engine and Scenario, in similar fashion to the way MMR works with Modeler and Translator. MOS manages object persistence. The architecture permits MOS to be capable of distributed operation, just like MMR, although this is not our focus in MOOSE.

4 MOOSE MODEL REPOSITORY (MMR) AND DISTRIBUTED OPERATION

There are two kinds of distributed operation to consider: one is where model definitions are distributed, with some classes defined here, others there; the second is where model execution proceeds as a distributed simulation, executing simultaneously on a number of hosts, with one object instantiated here, another there. The MOOSE architecture supports both kinds of distributed operation; the present implementation supports distributing definition of multimodels, as this is our primary research focus.

The MMR originated in a perceived need which arose in the stand-alone version of MOOSE to unburden the Conceptual Modeler in the MOOSE HCI from maintaining complex structures and relations among classes, objects, attributes, methods, and parameters. Originally, the model definition provided as

4

output of the HCI was a set of flat text files, similar in some ways to the HTML (hypertext mark up language) now ubiquitous on the web. We had already developed in the MOOSE Translator a capability to read and parse this model definition and build the aforementioned structures and relations, so it was a relatively simple matter to reuse this code, and add a sockets-based (TCP) communications layer. This effort not only succeeded, it also paved the way for extending the horizon of MOOSE from stand-alone system to web operation. Along the way, we kept the flat file format we had designed, and thus preserved the capability to load the MMR from one or more sets of flat files describing any numbers of previously-constructed models. This capability now makes it easy for an MMR to import model definitions created or modified by hand, which are easy to handle, transmit, and modify because of the flat text file format. We back up the MMR into this format; we dump model definitions into this format. While the format can be readily machine generated, it is also amenable (just like HTML) to being edited by hand with one's favorite text editor.

The MMR has a client/server architecture, with each MMR server maintaining a database of model definitions. MMR is in some ways is patterned after the CORBA (Common Object Request Broker Architecture) IR (Interface Repository) (Orfali, 1996). MMR as a MOOSE component does its part to support an overall model/view architecture, with multiple views being possible for a single model, and similarly, with multiple models being present in a single view. In the original stand-alone mode, the clients were the MOOSE Conceptual Modeler and the MOOSE Translator, with the Conceptual Modeler updating the MMR server, and the Translator querying it in order to emit Engine code in TTL. There was one MMR server, and it was co-located on the same host with the two aforementioned clients. In web-wide operating mode, MMR servers can be anywhere, can exist in any number, and can be shared; if each host has an MMR server, then the system offers greatest robustness in the face of network outage, but the architecture does not require it. MMR clients will usually be MOOSE HCI's and Translators; several HCI's located far from one another may collaborate to share and reuse model components; or, several Translators located far apart and unaware of one another's existence can be interested in using the same model or component definition. However, it is also part of the plan to expose an interface for other clients, which may be programs of any kind, including perhaps web browsers or other programs. This open architecture invites use of distributed model definitions outside of MOOSE to broader realms, as a more general object-
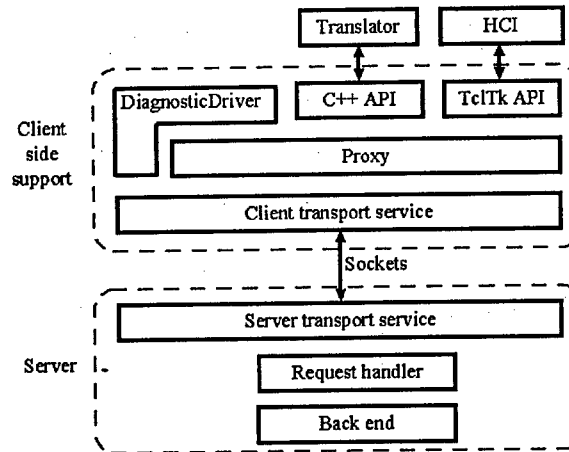

Figure 3: MOOSE Model Repository (MMR) Internals; Client above, Server below, each surrounded with dashed line; detail in accompanying text

oriented application framework. Time will tell if this idea will gain acceptance. The underlying MMR design is independent of whether MOOSE operates in stand-alone mode, or with clients and servers in any number and located far apart.

We now examine the MMR architecture which appears in Figure 3. Clients communicate with MMR using client side support. Two API's are shown: one for C++ code and one for TclTk, which support our Translator and HCI, respectively. Other API's are possible, should support for client code in other languages be needed. The client side support is layered as shown. Presently, our client side support is relatively thin. The diagnostic driver, providing built in support for test and development, is GUI-based, and allows developers and system maintenance technicians to operate the interface to the MMR server without a client program, permitting tests of Proxy and Client transport layers of client side support, as well as all of the server. The client communicates with the server using sockets, which are supported in both our platforms of choice (Windows NT and Solaris Unix), enabling client and/or server to be positioned on either platform with complete transparency. Sockets work whether clients and server are located on the same or different hosts. Client transport service is written in TclTk in a style which applies the OO principles available (encapsulation and information hiding). Server transport service is written in C++ with class names such as *Sockets*, *Hosts*, and *Circuits*. Proxy's counterpart on the server side is Request handler. Proxy and Request Handler work together. To the extent that we want to stage or cache information on the client side, this is hidden within Proxy. As previously stated our intent is a thin
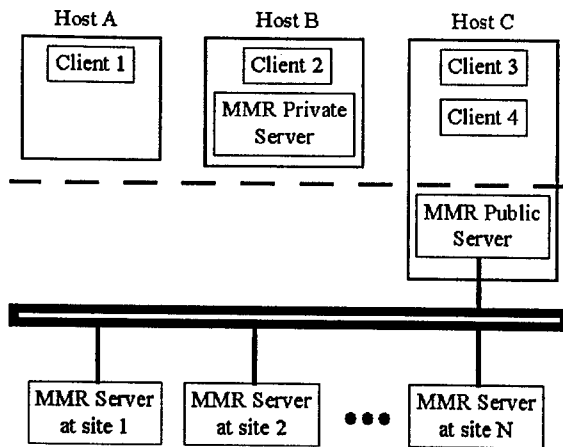
5

Host A　　　　　Host B　　　　　Host C

```
┌─────────┐   ┌──────────┐   ┌──────────┐
│ Client 1│   │ Client 2 │   │ Client 3 │
│         │   ├──────────┤   ├──────────┤
│         │   │MMR Private│  │ Client 4 │
│         │   │  Server  │   │          │
└─────────┘   └──────────┘   └──────────┘
─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─┌──────────┐
                           │MMR Public│
                           │  Server  │
                           └──────────┘
═══════════════════════════════════════
┌─────────┐  ┌─────────┐      ┌─────────┐
│MMR Server│ │MMR Server│ ••• │MMR Server│
│ at site 1│ │ at site 2│     │ at site N│
└─────────┘  └─────────┘      └─────────┘
```

Figure. 4: MOOSE Model Repository (MMR) as deployed on the Web; Dashed line is Firewall, above which is an Intranet; Heavy double line represents the Internet

client, but the presence of Proxy provides the ability to "thicken" the client side in the interest of performance, should that become necessary. Finally, the Back end provides data structures, linkage, and relations for classes, objects, attributes, methods, parameters, aggregation, association, containment, generalization, specialization, and inheritance; in short, the things one needs to know about a conceptual model.

Server Transport Service incorporates the initial sequence: create socket, set nonblocking, bind, and listen. Then periodically two activities are performed: accept'ing new connection requests, and servicing requests on existing connections, with priority given to the latter, and round-robin service policy. A dynamically-allocated self-expanding list of virtual circuits (connections) is maintained, so that an MMR Server can maintain any number of conversations with any number of clients and keep them all separate. Client Transport Service functions with send/receive pairs. Its receive is nonblocking; when there is no reply, the code is able to distinguish *end of file* from *no data yet*. This permits a client to retrieve long multi-message responses, and never to block. An interesting example of code reuse of the Client and Server Transport Services is this code also serves to synchronize Scenario and Engine, with Client Transport Service embedded into Scenario and Server Transport Service included in Engine runtime support.

Having examined MMR internal architecture, we now turn to two external views of MMR: first, the original stand-alone MOOSE which runs all processes on one host; second, distributed MOOSE which permits any number of MMR clients, any number of

MMR servers, and located on an arbitrary collection of hosts. The first view appears in Figure 2, and is relatively simple, where the MMR clients are the (conceptual) Modeler and the Translator as discussed above. The second view appears in Figure 4, and to this we now turn our attention. Above the dashed line appear three hosts, connected in an intranet. The dashed line is a firewall. A random collection of four client applications are shown; typically, these are instanmces of MOOSE (Conceptual) Modeler and Translator. Also above the dashed line firewall is an MMR private server, which is accessible to all clients in the intranet but not to any clients outside (below the dashed line). Just below the dashed line firewall appears an MMR public server. This server is accessible not only to clients above the dashed line but also to clients throughout the web. The heavy double line represents the internet and TCP/IP MMR protocol. Several distant MMR servers are shown at various web sites. Specifically, suppose that Client 1 is an instance of (Conceptual) Modeler which is building a model some of whose components are stored locally, in either the private or public server, with other components located at the MMR at site 1 and at the MMR at site 2. Client 1 is able to construct its large model from the various small ones transparently with respect to the location of the components. The illusion that the model definition is all stored locally is maintained by cooperation among the MMR client-side support services attached to Client 1 on Host A, the MMR Public Server on Host C, and the (distant) MMR Servers at sites 1 and 2.

Present plans call for the MMR protocol to be identical to the format already in use for the stand-alone version of MOOSE; this format appears in the flat text files which describe MOOSE conceptual models, and is HTML-like in the sense that it can be inspected and modified with almost any text editor, to facilitate diagnostic work as well as customization. Since the web is a network of multimedia documents, we propose a way of integrating the MMR with the web. This is done by a simple mechanism: permitting an object attribute to be of type *URL*, a class whose instances are URL's. Thus, documentation is an attribute of an object and within a web document, a conceptual model may be inserted as a basic URL type, e.g., *model*. Accordingly, to retrieve a conceptual model of a six-cylinder automobile engine from Detroit, the following hypothetical URL would be accessed: *model://models.gm.com/eng6cyl.mod*. This permits a tightly-coupled, interwoven effect between the web and a MOOSE conceptual model. MMR Servers will support this proposed framework when it is available; in the interim, they can communicate with TCP/IP, until there is demand to implement the

proposed mechanism.

5 CONCLUSIONS AND FUTURE DIRECTIONS

To date MOOSE has fulfilled each promise we had for its capabilities. We are gratified that OOPM has provided both a sound theoretical footing as well as a guide for our intuition as we develop MOOSE. Several research projects (e.g., a study of the Everglades ecosystem) are planning to work with MOOSE, and this fall students will use MOOSE in homework and projects for the graduate course in Simulation Principles at University of Florida, providing what is certain to be valuable feedback.

Distributed web-based operation is leading in new directions. Distributed operation questions include (1) how to categorize and locate components for reuse, (2) whether dynamic binding is the most appropriate binding time for component definitions, (3) how scalable the MMR will turn out to be, (4) what relation if any will exist between MOOSE, CORBA, and DCOM, and (5) how successful will be our approach to embedding legacy code as MOOSE models. Other questions include (6) whether Java will displace TclTk as primary language for MOOSE HCI's, (7) how to apply distinctions with greater sophistication among the relations containment, usage, composition, and association, (8) how best to extend the existing MOOSE repertoire for dealing with collections of objects to make it better serve model authors' needs, and (9) how to make Scenario's visualizer as generic as the rest of the model definition.

ACKNOWLEDGMENTS

REFERENCES

Berzins, V., M. Gray, and D. Naumann. 1986. Abstraction Based Software Development. *Communications of the ACM.* 29 (5): 402–415.

Booch, G. 1994. *Object-Oriented Analysis and Design with applications*, 2nd ed. Reading, Massachusetts: Addison-Wesley.

Cubert, R. M. and P. A. Fishwick. 1997a. MOOSE: architecture of an object-oriented multimodeling simulation system. In *Proceedings of SPIE – Society of Photo-optical Instrumentation Engineers; Enabling Technology for Simulation Science*, ed. A. F. Sisti, 3083:78–88. Bellingham, Washington: Society of Photo-optical Instrumentation Engineers.

Cubert, R. M. and P. A. Fishwick. 1997b. MOOSE: an object-oriented multimodeling and simulation application framework. To appear in *Object-Oriented Application Frameworks*, ed. M. Fayad, D. C. Schmidt, and R. Johnson. New York: John Wiley and Sons.

Fishwick, P. A. 1988. The Role of Process Abstraction in Simulation. *IEEE Transactions on Systems, Man and Cybernetics.* 18 (1): 18–39.

Fishwick, P. A. 1989. Abstraction Level Traversal in Hierarchical Modeling. *Modeling and Simulation Methodology: Knowledge Systems Paradigms*, Zeigler, B. P., M. Elzas, and T. Oren, eds. Elsevier North Holland. 393–429.

Fishwick, P. A. 1995. *Simulation Model Design and Execution : Building Digital Worlds.* Englewood Cliffs, New Jersey: Prentice-Hall.

Fishwick, P. A. 1996. Extending Object Oriented Design for Physical Modeling. *ACM Transactions on Modeling and Computer Simulation.* Submitted July 1996.

Fishwick, P. A. and K. Lee. 1996. Two Methods for Exploiting Abstraction in Systems. *AI, Simulation and Planning in High Autonomy Systems.* 257–264.

Fishwick, P. A. 1997. A Visual Object-Oriented Multimodeling Design Approach for Physical Modeling. Submitted April 1997 to *ACM Transactions on Modeling and Computer Simulation.*

Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM.* 33 (10):31–53.

Lin, Y. B. and P. A. Fishwick. 1996. Asynchronous Parallel Discrete Event Simulation. *IEEE Transactions on Systems, Man and Cybernetics.* 26 (4):397–412.

Orfali, R., D. Harkey, and J. Edwards. 1996. *The Essential Distributed Objects Survival Guide.* New

York : John Wiley and Sons.

Rumbaugh, J., M. Blaha, W. Premerlani, E. Frederick, and W. Lorenson. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall.

AUTHOR BIOGRAPHIES

ROBERT M. CUBERT is a Ph.D. student in the Department of Computer and Information Science and Engineering at University of Florida. His research interest is distributed object-oriented modeling and simulation. He holds BS degrees in EE from MIT and in Zoology from University of Oklahoma, and an MS in Computer Science from University of Oklahoma. He spent 3 years on Computer Science faculty at California State University, Sacramento, and has a decade of industry experience writing software for realtime control systems and communications.

PAUL A. FISHWICK is an associate professor in the Department of Computer and Information Sciences at the University of Florida. He received the BS in Mathematics from the Pennsylvania State University, MS in Applied Science from the College of William and Mary, and PhD in Computer and Information Science from the University of Pennsylvania in 1986. He also has six years of industrial/government production and research experience working at Newport News Shipbuilding and Dry Dock Co. (doing CAD/CAM parts definition research) and at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a senior member of the IEEE and the Society for Computer Simulation. He is also a member of the IEEE Society for Systems, Man and Cybernetics, ACM and AAAI. Dr. Fishwick founded the comp.simulation Internet news group (Simulation Digest) in 1987, which now serves over 15,000 subscribers. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990) and he is on the editorial boards of several journals including the ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics, The Transactions of the Society for Computer Simulation, International Journal of Computer Simulation, and the Journal of Systems Engineering.

MOOSE: an object-oriented multimodeling and simulation application framework .

Robert M. Cubert and Paul A. Fishwick

Department of Computer & Information Science and Engineering
CSE Room 301
University of Florida
Gainesville, FL 32611-6120

ABSTRACT

MOOSE (Multimodel Object Oriented Simulation Environment) is an application framework for modeling and simulation, under development at University of Florida, based on Object Oriented Physical Modeling (OOPM). OOPM extends object-oriented program design with visualization and a definition of system modeling that reinforces the relation of *model* to *program*. OOPM is a natural mechanism for modeling large-scale systems, and facilitates effective integration of disparate pieces of code into one simulation. Components of MOOSE are Human Computer Interface (HCI), Library, and Back End. HCI interacts with model author via graphical user interface (GUI) which captures model design, controls model execution, and provides output visualization. Library has a model repository and object store facilitating collaborative and distributed model definitions, and managing object persistence. The Back End automatically converts a model definition to a complete simulation program in some Translator Target Language (TTL), presently C++, then compiles and links the program it wrote, adding runtime support, and creating an executable program which runs under control of the HCI to provide model execution. Dynamic model types include Finite State Machine, Functional Block Model, Equational Constraint model, and Rule-based Model; alternatively, model authors may create their own C++ code methods; model types may be freely combined through multimodeling, which glues together models of same or different types, produced during model refinement, reflecting various abstraction perspectives, to adjust model fidelity during development and during model execution. Underlying multimodeling is "Block" as fundamental object. Every model is built from Blocks, expressed in a Modeling Assembly Language.

Keywords: Simulation, Multimodel, Object-Oriented Modeling, Model Abstraction, Object Oriented Physical Modeling, Visualization, Application Framework

1. INTRODUCTION

MOOSE is an acronym for "Multimodel Object Oriented Simulation Environment", a modeling and simulation enabling tool under development at University of Florida. MOOSE is an implementation of OOPM (Object Oriented Physical Modeling),[1] an approach to modeling and simulation which promises not only to tightly couple a model's human author into the evolving modeling and simulation process through an intuitive HCI (human computer interface), but also to help a model author to perform any or all of the following: (1) to think clearly about, to better understand, or to elucidate a model; (2) to participate in a collaborative modeling effort; (3) to repeatedly and painlessly refine a model as required, in order to achieve adequate fidelity at minimal development cost; (4) to painlessly build large models out of existing working smaller models; (5) to start from a conceptual model which is intuitively clear to domain experts, and to unambiguously and automatically convert this to a simulation program; (6) to create or change a simulation program without being a programmer; (7) to perform simulation model execution and to present simulation results in a meaningful way so as to facilitate the other objectives above.

In some cases model design, without model execution, suffices to achieve the model author's objectives, which may be to learn about or better understand a phenomenon or system, or to communicate about such a system with one's colleagues. This purpose is *per se* justification for the development of MOOSE. But usually a model author wishes not only to design a model but also to construct a simulation program to perform model execution, for reasons which include : (1) to empirically validate the model based on observed behavior; (2) to select or adjust various parameters and values and observe their effect; (3) to measure performance; (4) to gauge model fidelity and assess its adequacy.

In prevalent practice, a model author makes what is known as a *conceptual model*, often similar to a "blackboard picture" with annotations, and uses this model to describe to one or more programmers detailed requirements for a simulation program to be written, based on the conceptual model. Programmers then write a program, *but there is not necessarily a relation between the conceptual model and the program subsequently produced.* MOOSE offers to improve this situation: MOOSE assists the model author with constructing the conceptual model, and then builds a simulation program in an unambiguous way from the conceptual model. MOOSE thus provides a *mapping* from conceptual model to simulation program. Advantages include: (1) built-in model validation[2]; (2) partial automation of the development process, allowing model authors and programmers to focus on the difficult, rather than on the tedious; (3) easier accommodation to change, leading to a view of change as acceptable instead of as a threat; (4) reducing the response time associated with system development, allowing the model author to effectively drive the development process.

The amount of detail in a model reflects the model author's abstraction perspective.[3] Refinement to greater detail is used to obtain model fidelity that is adequate in the eyes of the model author, from a given abstraction perspective,[4] and with certain objectives for the model or simulation to meet.[5] MOOSE addresses this area with *multimodeling*, an approach which glues together models of the same or different types, produced during the activity of model refinement, and reflecting various abstraction perspectives.[6] Refinement can be adjustable during model execution as well as during model design. The pieces that are put together to form a model, such as described above, are *dynamic models*. Dynamic model types supported include Finite State Machine (FSM), Functional Block Model (FBM), Equational Constraint Model (EQN), and Rule-based Model (RBM); alternatively, users may create their own C++ "code models"; model types may be freely combined. The dynamic model types implemented so far form a popular collection of approaches used in simulation.[7] Additional dynamic model types are certainly in order and will likely be added to the MOOSE repertoire.

Reuse of one's own previous work, as well as by one model author of the work of others, is encouraged by availability of model repositories. These form a resource of growing value as MOOSE matures. For example, the "boiling water" model,[7] is an FSM multimodel, part of which is shown in Fig. 6. Later, we implemented a model of Robert Fulton's steamship,[8] whose FBM appears in Fig. 5. When the Fulton model was built, the boiling water model's *Pot* re-emerged as the *Boiler* of the steamship. Yet, an application framework is more than just a class library. In an application framework, classes from the library are related in such a way that a class is not used in isolation but within a design encouraged and supported by the framework. The MOOSE Model Repository (MMR) is aptly named because it is not just a class library; as a model repository, it stores not only a collection of classes available for reuse, but also the design which relates those classes as to how they play together within the geometry and dynamics of a particular model. This enables support for one of Booch's five attributes of a complex system (p.13): "A complex system that works is invariably found to have evolved from a simpler system that worked .... A complex system designed from scratch never works and cannot be patched up to make it work.". Using MMR, model authors can start from some piece of their overall system that happens to appeal to them intuitively. When several such pieces are working, they may be combined into a more-complex (working) system.

Components of MOOSE fall into three groups: Human Computer Interface (HCI), Library, and Back End. The HCI is comprised of two components: Modeler and Scenario. *Modeler* interacts with the human model author via graphical user interface (GUI) to construct the model. In simulation parlance, this is *model design*. Modeler relies on the Library (discussed below) to store model definitions. *Scenario* is a visualization enabler employing a GUI. Scenario activates and initializes simulation model execution (which we call Engine) at the behest of user (who may or may not be the original model author). Scenario maintains synchronous interaction with Engine, displaying Engine output in a form meaningful to user, optionally allowing user to interact with Engine, including modifying simulation parameters and changing the rate of simulation progress. The Back End has two components: Translator and Engine. *Translator* is a bridge between model design and model execution: Translator reads from the Library a language-neutral model definition produced by Modeler, and emits a complete computer program for the model, in Translator Target Language (TTL). Presently MOOSE TTL is C++; potentially, TTL can be Java or another language. This simulation program emitted by Translator is called *Engine*. Its source language is TTL, presently C++. Once compiled and linked with runtime support, the Engine executable program is activated under control of Scenario to perform model execution. The Library has two components: *MOOSE Model Repository (MMR)* and *MOOSE Object Store (MOS)*. MOS holds object data and MMR holds object meta-data. MMR keeps track of models as they are being built. MMR servers provide a database management system (DBMS) for model definitions.

MMR clients work with Modeler and Translator to define and use model definitions. Models and model components created by other model authors (or the same model author previously) are available for browsing, inclusion, and/or reuse. Base classes such as sets for modeling collections and popular geometries for spatial models are available to the model author. An MMR client can simultaneously maintain conversations with several MMR servers, each on a different machine, which permits model definitions to be distributed. An MMR Server can simultaneously maintain conversations with several MMR clients, on the same or different hosts, which permits collaboration on model development. MOS does for objects much of what MMR does for models. MOS works with Engine and Scenario, in similar fashion to the way MMR works with Modeler and Translator. MOS manages object persistence. Although the architecture permits MOS to be capable of distributed operation, just like MMR, this is not our present focus in MOOSE; thus, MOS operates in support of model execution on a single host only at this time.

There are two kinds of distributed operation to consider: one is where model definitions are distributed, with some classes defined here, others there; the second is where model execution proceeds as a distributed simulation, executing simultaneously on a number of hosts, with one object instantiated here, another there. The MOOSE architecture supports both kinds of distributed operation. The present implementation supports distributing definition of multi-models, as this is our primary research focus.

The balance of this chapter is organized as follows : section 2 explains how an Object Oriented approach is used by MOOSE to capture the geometry and dynamics of a model; section 3 explains how MOOSE uses multimodels to facilitate model refinement to achieve appropriate levels of model fidelity; section 4 describes the components of MOOSE in some detail and how they interact; section 5 covers some important MOOSE internal classes, as well as our chosen platforms; section 6 is our conclusions and plans.

2. AN OBJECT-ORIENTED APPROACH TO INTEGRATING MODEL GEOMETRY AND DYNAMICS

2.1. An Object Oriented Approach

MOOSE is the implementation of a simulation environment built using the OOPM design philosophy. Classes and objects in the digital world being built correspond to classes and objects in the real world being modeled. This
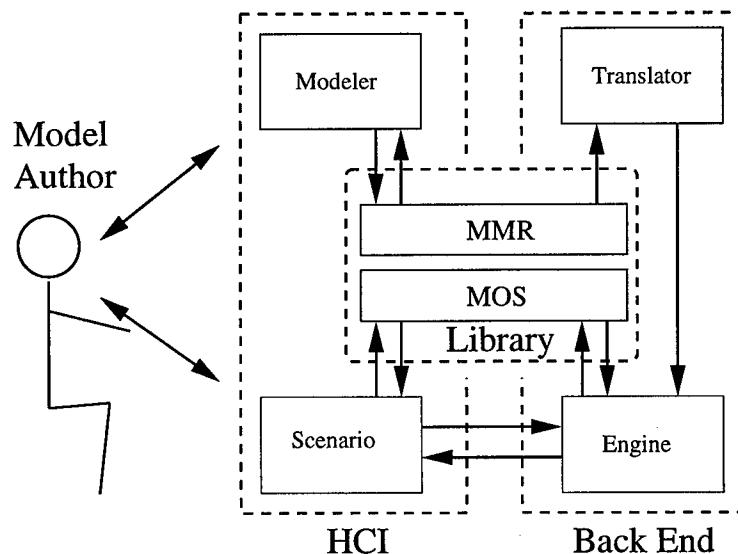
Figure 1. The three components of MOOSE (HCI, Library, and Back End) are shown outlined with dashed line boxes. Parts within each component are shown outlined with solid boxes: within HCI (Human Computer Interface) appear Modeler and Scenario; within Back End appear Translator and Engine; within Library appear MMR (MOOSE Model Repository) and MOS (MOOSE Object Store). Principal interactions are shown with arrows. The most important element is the model author, who appears at left interacting with both parts of the HCI.

approach has been found to not only be intuitively appealing to model authors, but also to be both effective and efficient at capturing the elements of meaning which must be represented in the model.[9] Dynamic models are an extension to "OO method"; static models, in the form of Abstract Data Types without dynamic behavior, are an extension to "OO attribute".

As (class and) object identification are performed, the model author is encouraged to explicitly recognize relations among classes. Most notable among these relations are **specialization** (or **generalization**) and **aggregation**. *Specialization* is the relationship of derived class (or subclass) to base class, such as "an orange is a *kind of* fruit"; *Generalization* is just the reverse, such as "Truck and Airplane are *kinds of* Vehicle". *Aggregation* comprises not one but several sometimes overlapping relations in the system being modeled, including *containment, composition, usage,* and *association,*[10–12] such as "a Car *has* Wheels", "the Moon *is made of* GreenCheese", "a Customer *uses* an automated teller machine (ATM)", and "a Teacher is *associated* with a University", respectively. Sometimes deciding which particular relation applies is a gray area; in any event, relations should not be examined in a vacuum but rather in the context of the model being built. **Containment:** "a Car contains an Engine". Litmus test: (1) does behavior of Car depend in fundamental way on Engine? Yes. (2) is Engine inside or part of or attached to Car? Yes, hence *containment.* **Composition:** "a Basket is composed of Straw". Contrast this with "a Basket contains Fruit"; hence, not containment, as the Straw forms the boundary not the contents. Constituent in a constructive sense. **Usage:** "a Person uses an ATM". Litmus test: (1) does behavior of Person depend in fundamental way on ATM: No. (2) is the ATM inside or permanently attached to the Person: no. Hence *usage.* **Association:** "a Car has a Manufacturer"[11] Litmus test: (1) does behavior of car depend in fundamental way on GM? No. (2) Is General Motors inside my Car? No. (3) Does the Car have an ongoing use for GM? No (although one can think about recalls, and/or suing GM for defects, but this is not within the ambit of Car behavior in most systems, because it's the owner who sues, not the Car which sues; and it's GM which recalls the Car, not vice versa).

Sometimes these distinctions cannot be drawn with certainty, and sometimes the model can be elucidated completely without deciding the issue; but discussing the nature of relations, thinking about them, and a reasonable amount of effort spent in attempting to categorize the aggregations within a model is often a useful exercise, because of the light it sheds as the discussion and debate unfold.

An example of drawing such distinctions is "containment by reference" *vs* "association by referential attribute".[11] Both are pointers so there is no implementation issue; but the difference is with regard to lifetimes: in the first case, the object contained by reference should live and die with the containing object; in the second case, the objects have independent lifetimes. This distinction may be useful for the model author to recognize.

Figure 2. Class relations depicted graphically as they would appear on a MOOSE HCI canvas to a model author. At left is aggregation; at right, generalization (viewed upwards) or specialization (viewed downwards). The aggregation relation is symbolized by the filled square; the specialization or generalization relation is symbolized by the filled circle. The small boxes just above each class in the aggregation specify *cardinality*, which is explained in the text. This aggregation, in words: a Teacher has many Students, has a Teaching style, and has an association with a University. The specialization, in words: Professor, Lecturer, and Adjunct are all kinds of Teacher.

12

Specialization and Generalization relation have been thoughtfully investigated.[13,14] Aggregation abounds in most models we have encountered, and we have found it to be of fundamental importance to the process of modeling. Aggregation has also received attention,[11,10] although less so than specialization / generalization. Aggregation has received, and continues to receive, keen scrutiny as we develop MOOSE under OOPM principles.

As MOOSE is requiring the model author to communicate relevant object identification and relations, it is building a conceptual model, which can be a handy representation for the model author. And this kind of "blackboard model" is often useful for one person involved in a project to communicate with his or her co-workers. Yet two other important things are going on: (1) as the model takes shape before his or her eyes, the model author often gains understanding, as represented in the aphorism "the best way to learn something is to have to explain it to someone else"; and, (2) a model description is being constructed which although independent of any programming language, is nonetheless unambiguously and automatically convertible to a simulation program in some programming language, e.g., C++, when the model author wishes to do so. Making explicit the classes and objects, and their relationships often sheds light on what is being modeled and surfaces questions and ambiguities which must be addressed to achieve the modeling or simulation objective. This is part of what is meant by tightly coupling the model author into the modeling and simulation development loop.

2.2. Attributes, Abstract Data Types, and Containers

Attributes are defined for each class. In addition to the "primitive" data types (integer, real, and string), MOOSE also permits arbitrary abstract data types (ADT) to be attributes. These ADT's are just classes like all the other classes in the model, and they play an important role in representing aggregation. We have several ways in MOOSE to represent aggregation as we create the constituent elements as attributes of the aggregate class. A decision as to the best representation rests on answers to questions such as whether the number of items of a constituent type is one, (e.g, a car *has* one steering wheel), or more than one (*e.g.* a car *has* four tires); whether the number of items of a constituent type elements is known in advance and fixed (number of cells on a checkerboard), or is inherently variable (a population of deer).

When specifying aggregation, the model author can choose from several *cardinality* alternatives for each aggregated class: **many** causes a container to be created, which holds contained objects of the aggregated class; a value such as **64** also causes a container to be created but additionally automatically populates the container with the designated number of contained anonymous objects; **A** causes an association, meaning a referential attribute, which is a reference (pointer) to a named object whose lifetime is independent of the lifetime of the object of the aggregating class; **V** indicates containment (value), which generates a value attribute within the aggregating class. When the cardinality of an aggregated class is one, an ADT attribute will be created in the aggregating class, but a choice remains between value and reference. In the first case an object of the aggregated class is created at the constructor time of the object of the aggregating class. This suggests the "lifetime test" as a decision criterion. If the aggregated object will have a lifetime which is independent of the lifetime of the aggregating object, then an association, represented by a reference (pointer), is in order. It is also possible for model author to choose a referential attribute when the lifetimes coincide, but then it will be model author's responsibility to manage object destruction. Because this is usually an unwelcome duty, value attributes should be chosen whenever possible.

A second criterion is the "name test". If the object of the aggregated class needs to be a named object created in another part of the model by the model author, then a referential attribute, also represented by a reference (pointer), is in order, irrespective of the lifetime issue. Yet we have found that named objects are the exception rather than the rule, because: names are often not a necessity; named objects force more work onto the model author; and, unnamed objects are still accessible as or within attributes of their aggregating class. We have found that MOOSE's ability to create these *anonymous objects* is quite useful. An example is a collection of 1000 individual models of free-roaming entities, such as polymers on a substrate or deer in a forest. In both cases the model author considers the objects a *fungible* collection, and has no need to provide each of the 1000 objects with unique names. MOOSE supports this with containers.

When there are multiple, and especially when there are an uncertain number of items of a kind, an abstract data type which is a *Container class* becomes the type of the attribute. This container class attribute holds elements of the contained type. For example, *Tires*, a tire container, might hold four tires, and this tire container is an attribute of the class *Car*. Alternatively, *DeerPop*, a deer container, might hold any number of deer, and this deer container is an

attribute of the class *Everglades*. Container classes have been found to be an effective way to represent an important aspect of aggregation. Provision is made in MOOSE for optional automatic population of containers at constructor time; and alternatively, to allow the model author an optional code fragment to append to the constructor to instead allow custom initialization of containers if required. Container classes can be specified directly by the model author, such as the DeerPop example above; or, are created automatically by MOOSE based on cardinality, such as the Tires example above, where the user actually only created the Tire class but mentioned that the cardinality is four. Containers have inherent behavior in MOOSE: they know how to send information to their contained objects, they know how to execute one or more methods of their contained objects; they know how to select a subset of their contained objects based on some criterion. This behavior is along lines discussed by Zeigler.[12]

Another aspect of aggregation is how to relate an attribute of an aggregating class with the corresponding attribute in its aggregated classes, when such correspondence exists. In contrast to a delegation function, which is a method of an aggregating class that simply passes through a method to an aggregated class, here we similarly have a method of an aggregating class that has the same name as a method of an aggregated class; however, here the problem is to invoke the corresponding method of every aggregated class and in some way transform the results into an overall result for the aggregating class. This problem necessarily includes the problem of a container obtaining such information from all its contained objects, which we have solved. The container problem is easier because all contained objects are homogeneous and can be dealt with in the same way. This suggests the solution, which is to derive all aggregated classes from a base class which has the desired functionality, and then package all the objects of the various aggregated classes into a single container whose contained class is the common base class, which has the requisite functionality. An example is biomass in an ecosystem simulation. A deer has a biomass which is its weight. Our deer population in the example above thus has a total biomass which is the sum of the weights of every individual deer. Moving to higher levels of aggregation, the Everglades has a biomass which is the sum of the biomasses of all populations in the model. Here the relation is *summation*, and the base class common to deer and fish and sawgrass has this functionality. While summation is a popular example, it is by no means unique; a model author is free to specify whatever functionality is appropriate to the model.

2.3. Capturing the Geometry of a Model

MOOSE is based on OOPM, and OOPM in turn has a number of tenets, two of its most important relate to geometry and dynamics. Geometry relates to space, and Dynamics has to do with temporal evolutions. We first discuss MOOSE model geometry. Geometry is represented by static models, in the form of Abstract Data Types without dynamic behavior, as an extension to "OO attribute". When a simulation involves a world where entities interact and evolve over a field, with the field often influenced and changed by the presence and activities of the entities, one usually thinks of model geometry in the conventional sense of defining properties of the space over which the field is defined and through which the entities move. This is certainly one form of model geometry, and one which MOOSE supports. An example of this kind of simulation is John H. Conway's board game "Life" ,[15,16] which has been implemented as a MOOSE model. A complete explanation of the game is not feasible here, but the interested reader can learn details from the references .[17] Summarizing the model, the *Game* is an aggregation of, or "has a", *Board* and *Rules*. Board in turn has a container *Cells* and a *BoardGeometry*. Cells container in turn contains many individual *Cell* objects. BoardGeometry maps the real location or identity of Cell objects in the Cells container onto 2-dimensional space. Rules tells the Game how to take each cell on the board from one tick of the simulation clock to the next (*e.g.*, birth, death). This illustrates that MOOSE can model *any* space by mapping elements of a container class of individual region objects onto a space of any specifiable complexity. In Fig. 3(a), a general framework is shown which applies to models we call "cellular". Such models are characterized as "field" models, and typically involve some kind of physical space with a certain geometry, a collection of entities that roam over the space, possibly interacting with one another and the space itself, and all of this evolving over time in accordance with some laws, which include initial and physical boundary conditions. An important subset of the "cellular" framework applies when the dimensionality is two. A framework for this subset which we term "landscape" is shown in Fig. 3(b). The Life model was constructed from the landscape framework. MOOSE considers geometry not only in the narrow conventional interpretation above; but also, in a broader sense, the space under consideration can be a space other than a physical space; rather, it can be a space over which classes and/or objects relate. MOOSE is capable of modeling this sort of geometry as well, through class definitions and relations among classes.
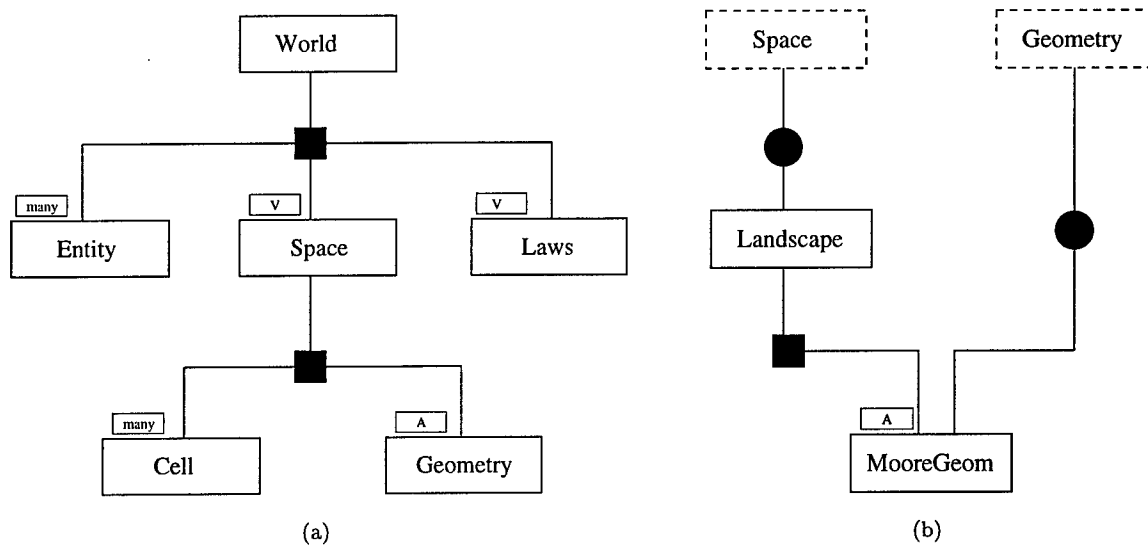
Figure 3. At the left is a MOOSE model that serves as a very general framework for a large variety of "field" models, which are characterized by a multi-dimensional space over which entities range, interacting with one another, and affecting the properties of the space, in accordance with initial and boundary conditions called Laws. The space is subdivided into cells, and has a geometry. At the right is a specialization of the cellular model, in which the space is two-dimensional and the geometry is that of a *Moore Neighborhood*, in which each cell has eight immediate neighbors. This model is encountered frequently, so it too has been given a name: "Landscape". A Landscape is a kind of Space. A Landscape has a Moore geometry, which is a kind of Geometry.

The focus on MOOSE so far is in support dynamic multimodels, but we envision equal support for static multimodels in the future. Static multimodels will define geometry and semantic networks appropriate for an object. For example, the hierarchy tree[18] is an example of a static model. To fully support static models, we will need to facilitate aggregation and inheritance in objects, as well as in classes.

2.4. Capturing Dynamic Behavior of a Model

Classes would be uninteresting indeed without methods. In MOOSE, these detailed aspects of every class may be readily added, changed, and removed, as part of model development, at any time. Dynamic behavior of the system is represented by *dynamic models*. Here MOOSE makes good its promise to the model author to be able to create or change a simulation program without being a programmer. MOOSE presently incorporates several kinds of dynamic model: FBM, FSM, EQN, and RBM, with others contemplated, such as Petri nets, and System Dynamics models. From this ensemble of popular and capable dynamic model types, the model author picks one or more dynamic model types to define methods of the classes of the model. Construction of each specific dynamic model typically involves drawing the kinds of "pictures" that people tend to make on the back of an envelope or a blackboard when informally describing a model to someone else. The MOOSE HCI facilitates these constructions: allowing the model author to specify components, connect components, provide inputs, outputs, conditions, and so forth.

2.4.1. Functional Block Model (FBM)

A Functional Block Model (FBM) is constructed when the model author so designates a method, *e.g.*, Mj, of some class, *e.g.*, Ci. The FBM editor enables the model author to construct the FBM for Ci::Mj(). Basic to an FBM is its blocks. The following are eligible to serve as blocks of the Ci::Mj FBM: (1) methods of Ci itself; (2) methods of any value attribute of Ci which is an abstract data type (ADT); (3) methods of any referential attribute of Ci which is an ADT; and (4) methods of other associated classes. The first two groups are bound at class declaration time. The last two groups require (or at least permit) dynamic binding.

15

The model author identifies each functional block of the FBM from the pool of eligible blocks. The blocks appear on the canvas as rectangles, like chips on a circuit board. Inputs and outputs of each block look like the pins on a chip. The next job of the model author is to connect the various pins with "traces", forming the topology of the FBM. Block outputs are connected to block inputs. FBM inputs are connected to block inputs. Block outputs are selected
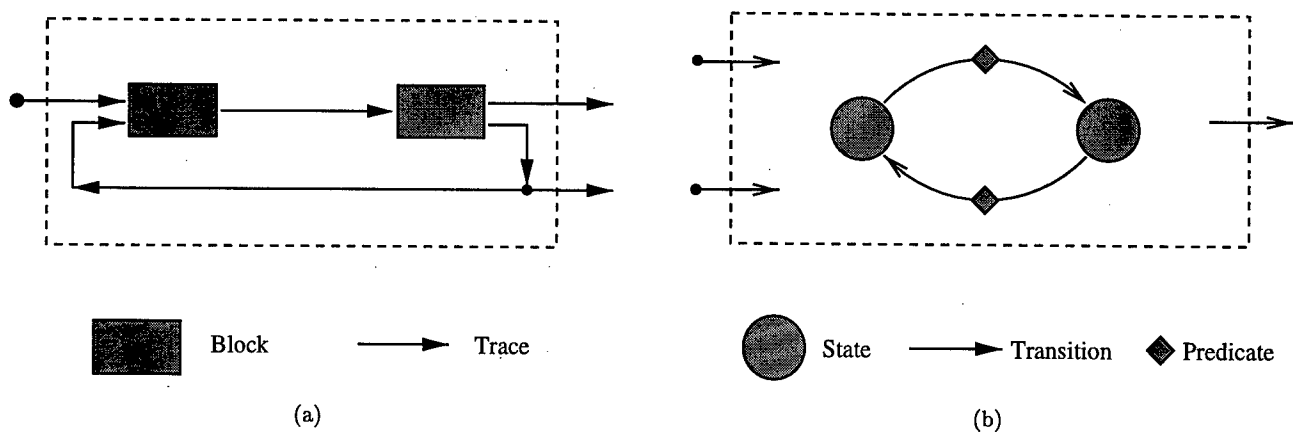

(a)   (b)

Figure 4. MOOSE Dynamic Models. On left, a Functional Block Model (FBM) On right, a Finite State Machine (FSM).
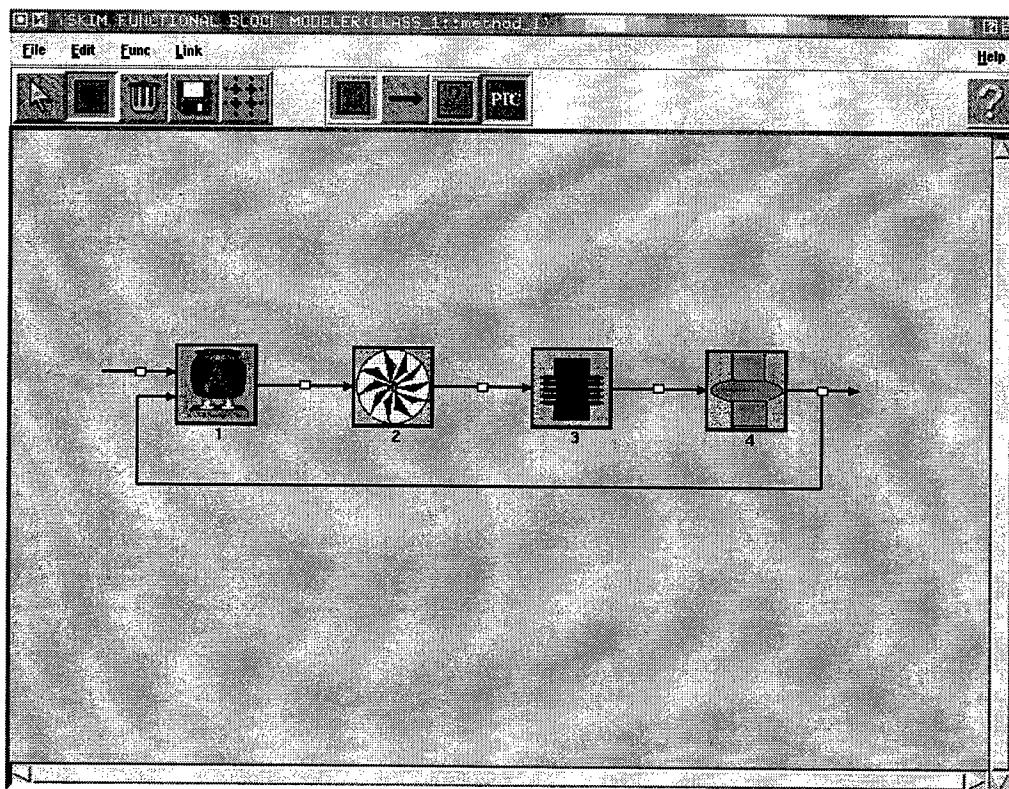
Figure 5. MOOSE HCI Modeler GUI Editor for FBM, showing part of *Fulton* steamship model, with functional blocks for Boiler, Turbine, Condenser, and Pump.

as outputs of the FBM. Cycles are permitted, in which case the value at one time step propagates to the next time step.

Several interesting collections are available to serve as blocks. One such collection is the "control" collection, consisting of Add, Subtract, Multiply, Divide, Integrate, Constant, PseudoRandom, and Accumulate. Another collection is the "queuing" collection, consisting of Source, Sink, Fork, Join, and Facility. Yet another collection is the "flowchart" collection, consisting of Begin, End, Decision, Process, and Auxiliary. The first collection is useful for building FBM's for control applications. The second collection is useful for simulating traditional queuing models. The third collection is useful for constructing models from flowcharts. FBM's can be constructed without using any of these collections, but these collections are available as ready-to-use components for those who are familiar with this modeling metaphor and wish to stay with it within the context of MOOSE.

2.4.2. Finite State Machine (FSM)

A Finite State Machine (FSM) is constructed when the model author so designates a method of some class. The FSM editor enables the model author to construct the FSM. An FSM consists of states. Eligible to appear as a state of an FSM are the same groups as are eligible to appear as blocks of an FBM, discussed above. After all states have been created and identified by the model author, they appear on the canvas as circles.

When states are in place, the model author constructs transitions between them. These transitions look like arrows. If the arrow points from state 1 to state 2 this corresponds to a transition of the FSM from state 1 to state 2. On each FSM transition, the model author places a predicate (logical expression). If the FSM is in a particular state and a predicate on one of its outbound transitions is true, then the FSM transitions to the state to which that transition points. A well-constructed FSM should have no more than one outgoing transition true at any time. If several such
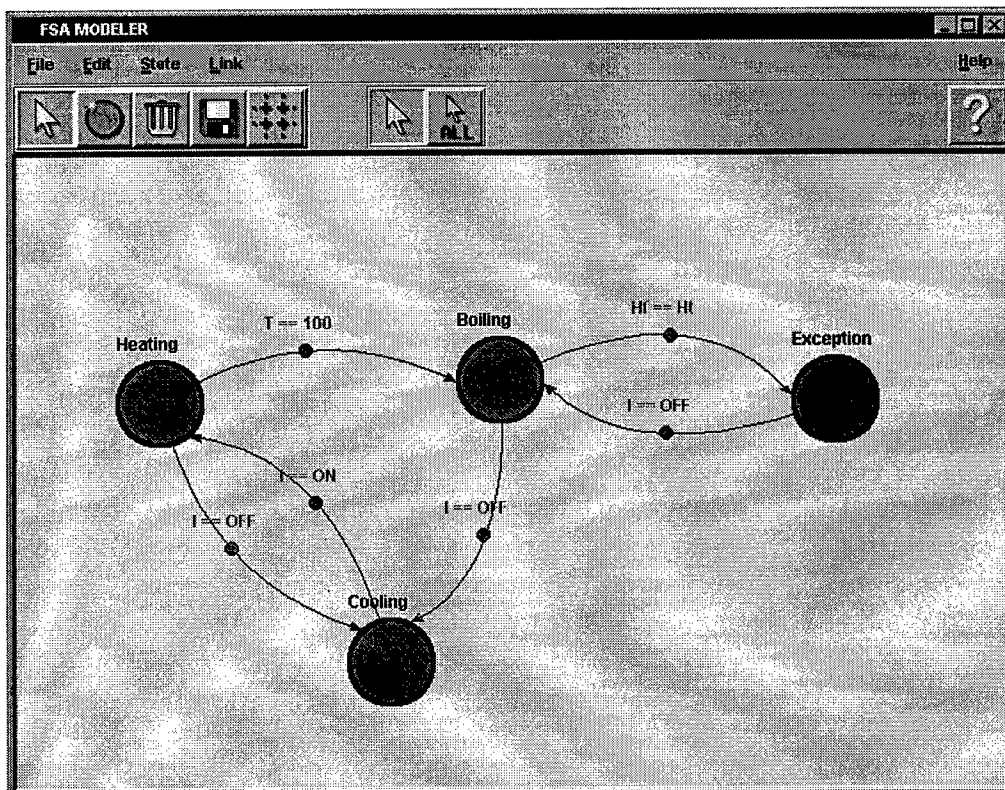
Figure 6. MOOSE HCI Modeler GUI Editor for FSM, showing part of *boiling water* model, with states for Heating, Cooling, Boiling, and Exception; predicates defining FSM state transitions appear as text near the tiny circles on arcs. Boiling water model is one component of Fulton steamship model: this FSM is actually inside the Boiler functional block of the FBM shown in the previous figure.

17

transitions are true in a MOOSE FSM, an arbitrary transition from among the transitions with true predicates is selected.

2.4.3. Rule Based Model (RBM)

A Rule Based Model (RBM) is constructed when the model author so designates a method of some class. The RBM editor enables the model author to construct the RBM. An RBM has a number of rules. Each rule is in the form of a a conditional expression: if *premise* then *consequence*. The RBM editor main window creates any number of rule templates in this form, and sets up each premise and consequence to allow the model author to choose any premise or any consequence to elaborate a new rule, or to change an existing rule. When the model author chooses a premise, he or she enters a premise-editing window, with various widgets that facilitate picking eligible items from lists, specifying relational and logical operators. A premise may be a simple logical expression or something more complicated; if the latter, then the premise becomes a block. The model author also defines each consequence, using a consequence-editing window. Each consequence is either a simple statement or something more complex; if the latter (as is usually the case), the consequence is a block. The ability to connect to blocks in this way fits RBM's into multimodeling hierarchies (to be discussed below).
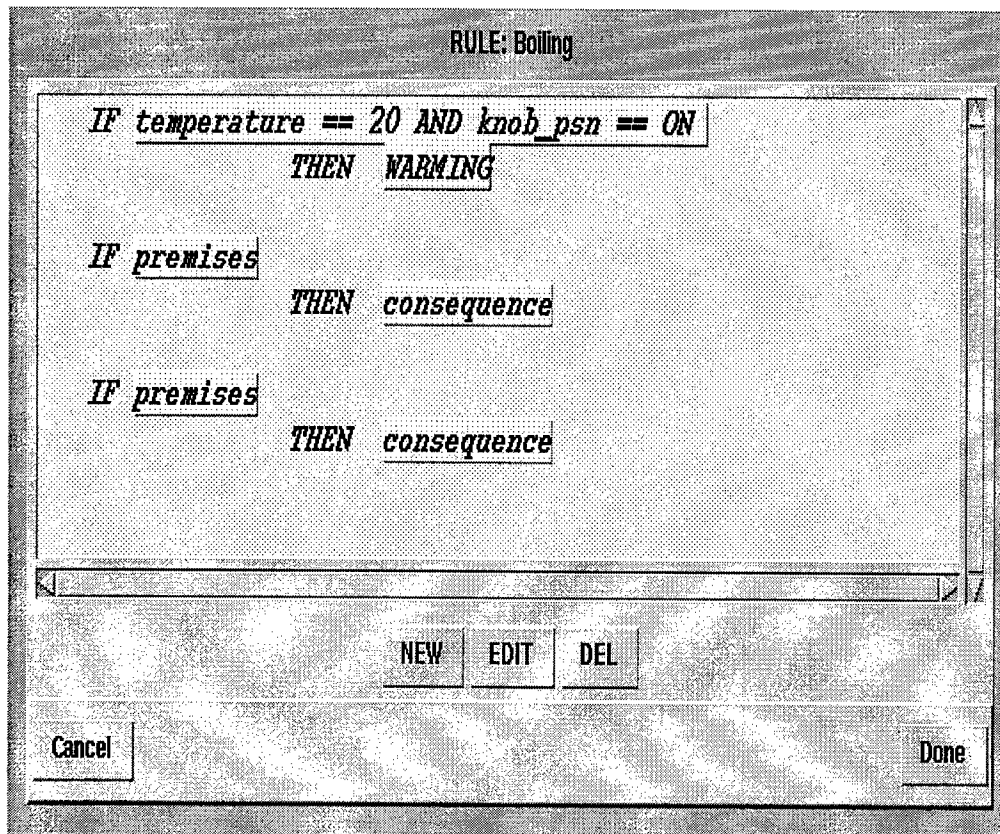
Figure 7. MOOSE HCI Modeler GUI Editor for RBM (Rule Based Model), showing the RBM editor main window. In this window, the model author has decided to construct three rules, which the editor has constructed as shown. The first of these rules has been turned from a template to an actual rule by the model author. The other two rule templates remain available. By clicking on a *premise*, the model author can edit the premise part of a rule in the premise-editing window (not shown); similarly, by clicking on a *consequence*, the model author can edit the consequence part of a rule in the consequence-editing window (also not shown). A premise can be a block. A consequence is always a block. Thus RBM's fit into multimodels just like other dynamic model types.

2.4.4. Equation Constraint Model (EQN)

An Equation Constraint Model (EQN) is constructed when the model author so designates a method of some class. The EQN editor enables the model author to construct the EQN model. A system of any number of nth order differential equations may be entered, using an intuitive syntax. Differential equations are represented using symbols such as x, x' for the first derivative of x, x" for the second derivative of x, and in general x followed by n single quote marks to denote the *nth* derivative of x. Several variables may be used. The output of the system may be any order derivative of any variable. If a variable used in the system of equations also happens to be an attribute of the class to which the EQN model belongs, then at the beginning of the computations of the EQN model at each time step, the EQN model is loaded with the value of that variable; and, when the computations are completed for that time step, the value is sent to that variable.

In addition to variables and their derivatives, a set of equations may contain (additive and multiplicative) parameters and input signals. Parameters may be attributes of the class to which the model belongs; or, they may be input parameters to the EQN method; or, they may be blocks, with eligibility the same as was set forth in detail for the FBM above. This will be discussed more fully when multimodeling is considered in a later section.

2.4.5. Code Methods

Although promising models without programming, MOOSE also tries to be tolerant and flexible. Thus, if none of the dynamic model types suits, the model author is free to write what are termed "code models" or "code methods". A code method is a function body written in the TTL used for the MOOSE system. Presently this language is C++. MOOSE design ideology suggests that code methods be the exception rather than the rule. Typical use of a code method is to provide that one small piece of some models that cannot be described using the available dynamic model types, and to rely on dynamic models for the rest, in a way that is analogous to construction of an Operating System kernel in a high level language, with just a few assembly-language routines where needed.

3. FACILITATING MODEL REFINEMENT

Constructing a model is almost always an iterative process, with model structure taking on a tree-like appearance. The broadest description of the model is like the root of a tree. One then typically decomposes this broad but nebulous description into subordinate parts, each part being a refinement of the model in the broad description statement. The result typically is a tree with some leaves near the root, and others farther "down". Thus the level in the tree is related to the level of abstraction which one associates with thinking about and describing the model.

To support the kind of heterogeneous model hierarchies shown abstractly in Fig. 8, we must ensure that our models are *closed under coupling*. In short, this suggests that the method of coupling one model component to another must be clearly defined. Two kinds of coupling exist: intralevel and interlevel. Intralevel coupling reflects model components coupled to one another in the same model. For example, one needs to specify rules of how Petri nets,
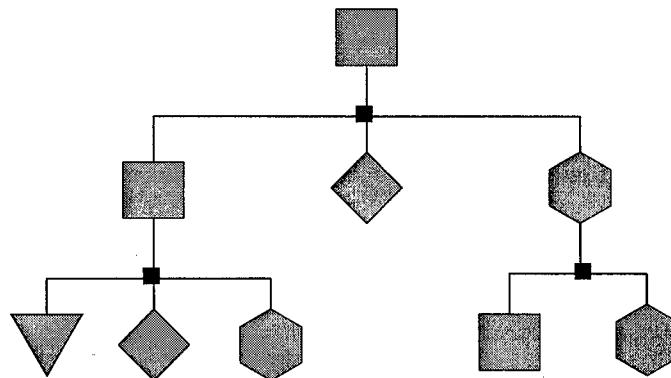

Figure 8. Multimodeling tree structure for model refinement. Selective refinements achieve required fidelity. Extensibility facilitates model development. Polygons above depict the heterogeneous nature of multimodeling: each type of polygon represents one type of dynamic model.

compartmental models and System Dynamics graphs are formed. With a System Dynamics graph, a rule of model building defines that any level has an input rate and an output rate. A more interesting case arises in interlevel coupling since we must ensure that we define rules as to how model components from one model can be refined into models of different types. Can a finite state machine state be refined into a Petri net, or can a functional block model contain finite state machines (FSM) inside blocks? What are the rules to guide this refinement? The rule for intralevel coupling is based on functional composition. The primitive of *function* with its *input* and *output* defines the coupling procedure in the following way. All models are encapsulated in a single function. Fig. 4 demonstrates this functional block encapsulation. This represents the outer shell to support interlevel coupling. Within a model there are *functional entry points*. These are inner shells where new models may be optionally inserted. Each model type has its own entry point defined differently. For example, for the model type "FSM", we may define each state to be of the form: $v(state) = f()$ where $f()$ is an arbitrary function and $v(state)$ defines the value of the attribute *state*. If *state* is not refined, then $f()$ returns the value of the state as a character string or integer. If *state* is refined, then $f()$ may be replaced by *any* function—whether this function is a dynamic model or a code method. The coupling approaches are defined in more detail by Fishwick.[19]

Resources are limited, and by this we mean both model development resources and simulation runtime resources. Thus one typically refines using a breadth-first approach, and this tree-like structure accordingly takes on an uneven shape, with some parts of the tree being of greater height, and others being of shorter height, reflecting the underlying decision criterion, which is to refine only as much as needed to achieve required levels of model fidelity. But knowing what is needed to achieve required levels of model fidelity often requires iterating through several model designs, and even measuring performance of the model execution. Multimodeling can be used in the development process, to conserve valuable development resources, including time, by limiting the depth of some subtrees; and to provide a top-down skeleton within which development may proceed. A rude shallow model can be run, and analysis can pinpoint those model subtrees where additional fidelity is needed. This is an adaptive mechanism to focus and guide development. The evolving model is thus its own prototype. It needn't be discarded, as in throw-away prototyping, nor does it suffer the chaos that often accompanies the "exploratory prototyping" or "exploratory programming" approach.[2]

Thus MOOSE provides facilities for multimodeling,[20,21] by which we mean model refinement into more detailed component models, reflecting a number of abstraction perspectives.[6] This is a very intuitive concept: most people multimodel without thinking about it; yet, they *do* benefit from encouragement in this direction, and especially from
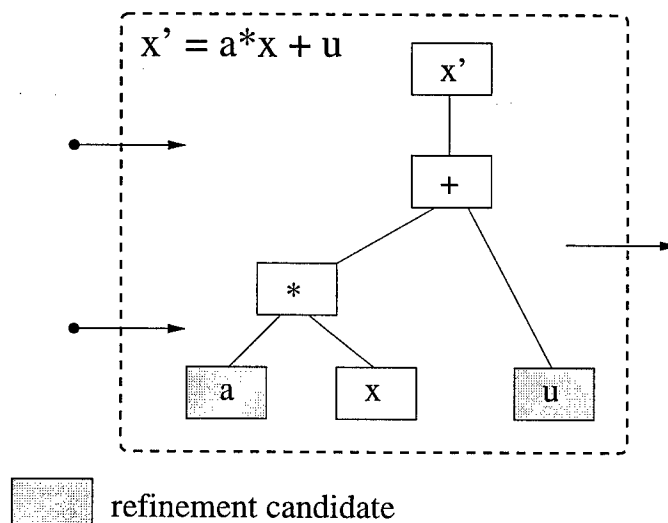

Figure 9. This figure illustrates a simplistic equation constraint model. The purpose is to indicate multimodeling in this dynamic model type. The rectangles representing parameters and inputs (a and u, respectively) are eligible to be refinement candidates; in other words, each of them may be block, which is a complete dynamic model in its own right.

automatic management of the resulting complexity. The multimodel definition is recursive: refinement proceeds as far as needed. By facilitating this kind of work and encouraging this kind of thinking, MOOSE contributes to management of the model: extending (or reducing) model refinement at any stage of the game. Typically, refinement is extended when fidelity is inadequate, and is reduced when the simulation takes too long to execute. Now having explained multimodeling in general, we proceed to a specific taxonomy. There are two perspectives from which one may look at multimodeling: time of binding and dynamic model type. Each perspective leads to a dichotomy. The overall result is a small taxonomy which will now be presented.

Multimodel dichotomy based on time of Binding: In a temporal sense, regarding the time at which the level of refinement is bound or fixed, MOOSE recognizes two kinds of multimodel: fixed structure and variable structure. *Fixed Structure Multimodels*: The evolution of model refinement that takes place over the model development life cycle results in a fixed structure multimodel; that is, we change the model's structure from time to time, but whenever we build a simulation program representing the model, we freeze model structure as of that time. Of course we can change it later and build a new simulation program, but the fixed structure multimodel persists until this is done. The second kind of multimodel is the *Variable Structure Multimodel*, and the MOOSE runtime environment supports this kind of multimodel too. A variable structure multimodel changes its refinement on the fly, in response to system constraints. A typical constraint is a realtime constraint on when the simulation must complete. Presently, MOOSE does not provide the executive logic which decides when to change refinement depth; but, given such logic, MOOSE has the capability to reconfigure model refinement on the fly. Others are presently working on providing this logic for MOOSE.[8]

Multimodel dichotomy based on type of Dynamic Models: When a model is refined, each level is usually described by one or more dynamic models. Each dynamic model is of some type, e.g., FSM. If all the dynamic models are of the same type, then the multimodel is *homogeneous*. If the dynamic models are of different types, then the multimodel is *heterogeneous*. In Fig. 8, for example, the multimodel depicted is heterogeneous.

4. THE COMPONENTS OF MOOSE

4.1. Introduction

MOOSE has six components, which fall into three groups: Human Computer Interface (HCI), Library, and Back End. Each group has two components. The HCI is subdivided into Modeler and Scenario. Modeler constructs a model; Scenario runs it. The Library is subdivided into MOOSE Model Repository (MMR) and MOOSE Object Store (MOS). MOS holds object data and MMR holds object meta-data. MMR keeps track of models as they are being built. MOS keeps track of objects as models execute. The Back End is subdivided into Translator and Engine. Translator converts a model definition to a program; Engine is that program.

4.2. Modeler

The Modeler component of the MOOSE HCI interacts with the human model author via a graphical user interface (GUI) to construct a model. In simulation parlance, this is *model design*. Modeler relies on the MOOSE Model Repository (MMR, discussed below) to store model definitions as they are constructed, and also relies on MMR to provide reuse components developed elsewhere by others or earlier by the model author. The Modeler GUI is a composite: the "main" part defines classes and objects and relations among classes (aggregation and specialization or generalization) on one or more canvases. On the canvas, rectangles represent classes. These rectangles are joined by relations to form a tree, or, more generally, a graph, reflecting the relations in the system being modeled. Some models look cleaner if aggregations and specializations are kept on separate canvases; this is supported but not required. Similarly, some models are large enough that several canvases are needed to capture the representation.

Each class is a box which, when opened, reveals more information, and permits the model author to define the name of the class, its attributes, its methods, and its named objects. Within each method, the model author may specify input parameters and output parameters, as well as identifying which dynamic model type the method is to be. In addition to the "main" GUI presented above, there is a GUI editor for each dynamic model type, *i.e.*: the FSM

editor for finite state machines, the FBM editor for functional block models, the EQN editor for equation constraint models, and the RBM editor for rule-based models.

Besides dynamic models (FSM, FBM, EQN, and RBM), the model author may also select code methods: ordinary code methods (CODE), constructor (CSTR), and destructor (DSTR). For these code methods, MOOSE provides a text editor which permits the body of each such method to be coded in Translator Target Language (TTL, presently C++). Since MOOSE is not really in the text editor business, and its text editor is relatively primitive, the model author is free to use his or her favorite text editor to modify these code methods. Because the emphasis in MOOSE is on dynamic models, not code methods, reliance on code methods should be minimal in most models. But as MOOSE philosophy is to facilitate rather than dictate, the code methods are available as the model author sees fit to use them.

The flow of information between Modeler and the model author is definitely bidirectional. If the model author builds a model from scratch, then the information flow is from model author to Modeler. We expect this not to be the usual situation. When the model author goes shopping for reusable components to include, or a previous model to modify, then the information flows from Modeler to model author.

Central to Modeler's ability to flow information both ways is MOOSE Model Repository (MMR). MMR has a client/server architecture, and Modeler is one of its clients. Modeler communicates via pipes with an MMR proxy it starts as a "subprocess". An MMR proxy is thus dedicated to this one instance of Modeler. The MMR proxy responds to requests from Modeler, obtaining MMR services from one or more MMR servers, which may be local or remote. In this way, Modeler has access to model definitions developed earlier "here" or "elsewhere", for reuse. Thus small working models can become elements of the model under constructions; or, a large working model can

(a)   (b)
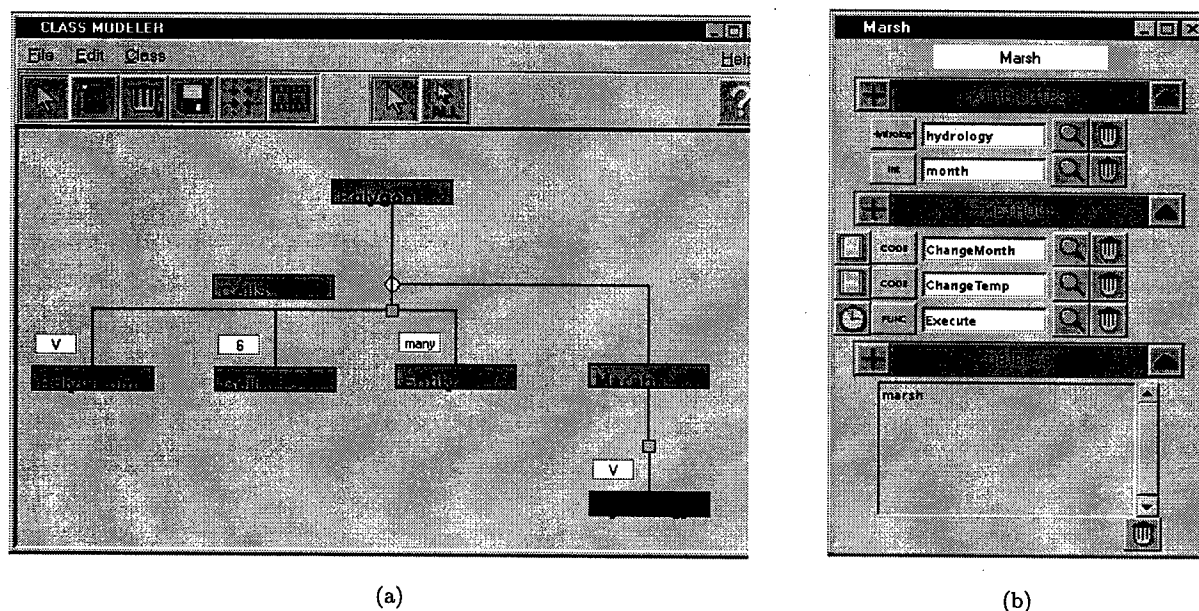
Figure 10. In this view of the MOOSE HCI, the Modeler GUI is shown for a landscape ecology model in which apple snails in Florida's Everglades are modeled as a collection of population models each within a spatial cell. Cells tessellate the polygon that represents the march environment of the Everglades. Apple snails are an important food of wading birds, and wading birds' behavior are studied using a *multimodel*, one piece of which will be the apple snail model. On the left, the Modeler canvas, showing classes and their relations. Aggregations: a Polygon has a geometry, some Cells, and many Entity's; a Marsh has a Hydrology. Specialization: a Marsh is a kind of Polygon. On the right, the view inside one of the classes is shown. This is available by double-clicking the class on the canvas. Information about attributes, methods, and instantiated objects appear.

be modified to a new purpose.

As Modeler receives information from the model author, it is retained not in Modeler itself, but in MMR. Modeler tells MMR whatever it learns of the model, and later queries MMR when it needs information to display the model. This allows Modeler to focus on its essential GUI model definition job rather than having to be in the DBMS business. A good example is when a named object is created in some class which happens to be derived from (a specialization of) one or more base classes. When the model author displays the object with the intention of viewing its methods, a recursive traversal is done "up" the generalization hierarchy, because methods of the object include not only the methods of its class but also the (public and protected) methods of its bases class(es). Modeler gains this knowledge from a query to MMR. All such methods come back to Modeler including their names and parameter lists. Thus the display includes the names and parameter lists of the methods of the object, as required, without Modeler having to maintain this information explicitly.

As a model author develops a model, one good mode of action is: develop a little, translate, and save. These steps are then repeated as the model grows. It is good practice to translate every so often to check for errors. If the change from the previous version is small, it facilitates localizing the source of such errors. If for any reason, the model author cannot resolve such errors, an option then exists to discard the latest change, reverting to the previous saved version. A second development mode is: develop a little, translate, execute, save. Again, these steps are repeated as the model grows. The procedure is similar to that above, except this time the model author has the opportunity not only to surface translation time errors, but also to verify the expected runtime behavior. Again, taking sufficiently small steps tends to improve productivity by helping to localize the source of errors, and allowing abandonment of a relatively small change that did not work as expected.

4.3. Translator

The Translator component of the MOOSE Back End uses a model definition from MMR to construct a simulation program in Translator Target Language or TTL. Our first TTL was C++, and this C++ Translator is in use presently. A second Translator is under development, with Java as its TTL.

Translator may be invoked either from Modeler or from Scenario. During development, it is most convenient for the model author to invoke Translator from Modeler. For production runs, it is easier for the user, who may not be the model author, to invoke Translator from Scenario. Translator's output, as previously mentioned, is a complete "Engine" program written in TTL (including indentation and comments). The C++ Translator specifically emits: *engine.h*, a header file consisting primarily of class declarations, and *engine.cpp*, a source file containing C++ translation of each dynamic model method and each code method, as well as code to invoke engine runtime support, and to synchronize with and accept commands from Scenario.

Many decisions are made in the course of deciding what code to emit and when. The heuristic we adopted is to move as much "intelligence" as possible out of Translator and into MMR. The effect of this is to make it easier to construct future Translators, as the same MMR serves them all. The cost in increased size and complexity of MMR appears to be justified: MMR has about 2/3 of the code and Translator has about 1/3, so this approach leverages Translator development by something like 3:1 through reuse.

4.4. MOOSE Model Repository (MMR)

The Moose Model Repository (MMR) component of the MOOSE Library holds object meta-data, such as class declarations, including declarations of attributes and methods, and class definitions, including method definitions. MMR keeps track of MOOSE models as they are being built. MMR also maintains collections of previous models and model parts for reuse.

MMR has a client/server architecture, and in some ways is patterned after the CORBA (Common Object Request Broker Architecture) IR (Interface Repository).[22] The MMR servers provide a database management system (DBMS) for model definitions. MMR clients work with Modeler and Translator to define and (re)use model definitions. Models and model components created by other model authors (or the same model author previously) are available for browsing, inclusion, and/or reuse. Base classes such as sets for modeling collections and popular geometries for spatial models are available to the model author. An MMR client can simultaneously maintain conversations with several MMR servers, each on a different machine, which permits model definitions to be distributed. An MMR

Server can simultaneously maintain conversations with several MMR clients, on the same or different hosts, which permits collaboration within an engineering workgroup on model development.

As was mentioned above, it is necessary at numerous points to analyze information before code can be emitted by Translator. Such analysis is performed by MMR as the data is entered. The result is typically stored in a data field, often as an *enum* type value or a boolean. Then in Translator, because the decision has already been made, complexity is typically reduced to a *switch* or *if* statement to carry out that decision. This makes MMR more than a DBMS: the model analysis aspect is an integral part of understanding a model definition sufficiently well to convert it automatically into a program. Here is an example, which occurs whenever a functional block model (FBM) appears as a method within a model. Each block of the FBM must be examined to ascertain whether it is (1) a method of the class containing the FBM, (2) a method of an ADT attribute of the class containing the FBM, or (3) a method of some other class. Each case is handled differently when code is emitted: a member method name, a method name qualified with the attribute name, or dynamic binding of a block from the model's context. MMR does this analysis and provides an enum type plus a boolean containing the decision, which allows Translator to effectuate the decision when code is emitted.

In addition to the normal mode of receiving model definition(s) from model author(s), MMR can also receive model definitions in another way: from text files. These files can be created using a text editor. Historically, such files were originally created by Modeler before MMR came into existence. These files now serve as a way to initialize or reload an MMR server.

The Modeler is connected to MMR via an interface that consists of pipes to a proxy process, and thence via TCP to an MMR Server. This architecture permits a proxy to maintain simultaneous conversations with several MMR Servers, some or all of which may be located elsewhere than the local machine. Additionally, the MMR Servers may converse with one another, to establish and maintain distributed definitions of models, based either on a hot link or a cached local copy of each distant component.

In the case of Translator and other C++ programs, access to MMR is provided by an API whose code is part of the Translator process itself, rather than a proxy. This is done for reasons of efficiency and simplicity, and does not compromise the architecture. Both the proxy and the API are "thin" programs, relying on the MMR Server, for two reasons: so that more code can be shared between them, and to offload issues of synchronization and concurrency to the MMR Server (where it "belongs").

MMR is presently a homegrown C++ creation intended as a proof of concept rather than for production use. Upgrade to "industrial strength" can be done in future without altering the architecture, as follows: the MMR Server can be replaced by a DBMS, either an OO DBMS, or an RDBMS with OO wrapper. Proxy and API will then be modified to make requests (queries and updates) to the new DBMS rather than to the original MMR Server. This brings to MOOSE the synchronization, locking, and security capabilities offered by commercial DBMS software. Importantly, Modeler and Translator never see the change and need not be modified.

4.5. Engine

The Engine component of the MOOSE Back End is generated by Translator. Translator emits Engine source code. It is then necessary to translate the Engine to create an executable (even with Java, into bytecode). In MOOSE this is done automatically under the covers using a "make" utility program; alternatively, Engine can be compiled and linked directly by a compiler such as Visual C++ or g++. At link time, a number of runtime support components are added from object libraries, the most important of which is *ooSim*.[23]

The ooSim event scheduling toolkit: All dynamic models are translated into C++ code which relies on the underlying event-scheduling of the *ooSim* dispatcher for propagating event chains. *ooSim* is an event-scheduling simulation queuing model toolkit which arose as an object oriented re-implementation and extension of the SimPack toolkit[7,24]; SimPack is, in turn, based on SMPL.[25] In addition to event scheduling, ooSim also provides numerous other forms of support, such as pseudo-random number generation. But it is the event scheduling that is ooSim's primary support role.

Engine source file contains code to initiate one or more event chains. These event chains propagate independently of one another, and the time step of each chain is independent of the time step of every other event chain. The event scheduler propagates each event chain until that event chain terminates itself, or until the simulation clock reaches

the overall time limit specified for the simulation in the model definition. In general, an event chain propagates by rescheduling a specific event routine which the model author identifies. This is accomplished by enabling the *auto_propagate* feature, which is done by default. However, it is also possible for the model author to disable *auto_propagate*, in which case the model itself may generate any number of event chains following any logic. This is an advanced feature which is recommended only to those who are familiar with event scheduling in ooSim and wish to (or need to) have the additional flexibility which manual event scheduling provides. Manual event scheduling is *not* required to get MOOSE models to run.

As the Engine runs, it executes one simulation event after another, driven by its underlying ooSim Future Event List (FEL). As an event executes, it may generate output on standard output (cout). All such output is presented to Scenario for possible use. See the Scenario subsection below for more detail. After executing each simulation event, Engine checks with Scenario for instructions and new parameter values. The relation between Engine and Scenario is thus inherently interactive and bidirectional. One such instruction permits Scenario to inject events into the FEL of an Engine as it is running. This is one feature necessary to support distributed execution of simulation models.

4.6. Scenario

The Scenario component of the MOOSE HCI is a visualization enabler employing a GUI. Scenario activates and initializes simulation model execution (which we call Engine) at the behest of user (who may or may not be the original model author). Scenario maintains synchronous bidirectional interaction with Engine. In the visualization role, Scenario displays Engine output in a form meaningful to user. In the controlling role, Scenario allows the user to interact with Engine, modifying simulation parameters and changing the rate of simulation progress.

Once the Engine executable has been built, it can be run as many times as desired, under auspices of Scenario. Scenario establishes a bidirectional pipe connecting it to Engine. The effect is to activate Engine, and to synchronize Engine's execution with that of Scenario, so that whatever Scenario writes to the pipe appears on the standard input
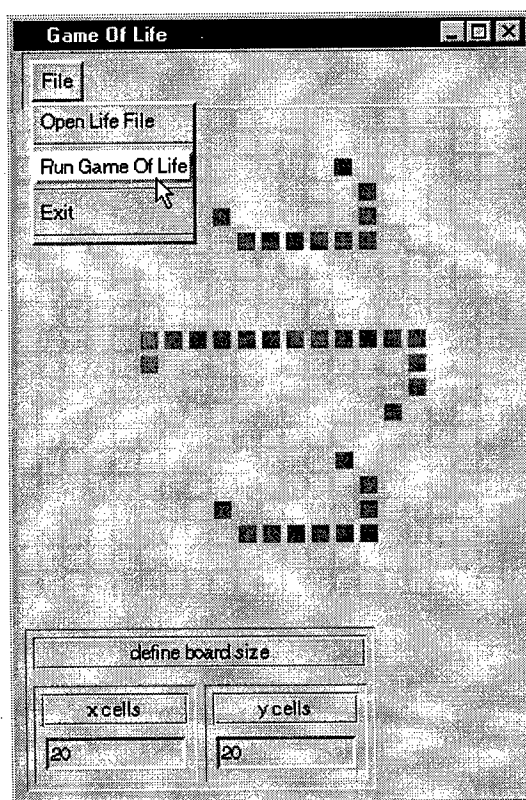
Figure 11. MOOSE Scenario GUI for Conway's game *Life*.

(cin) of Engine, and whatever Engine writes to standard output (cout) can be read from the pipe by Scenario. This interactive connection controls the real progress of Engine: Engine can be allowed to free-run, or can be made to single step through one event at a time (the default), or to run at any pace in between. As a separate feature, simulation clock time scales can be stretched or compressed. Both can be combined to generate animations with which the model author can interact. Things which ordinarily happen blindingly fast can be slowed down. The rate of progress can be adjusted to focus on parts of the simulation execution that are of particular interest.

The bane of simulation is output in the form of reams of computer printout. Scenario improves the situation with a GUI with which the user (who may not be model author) interacts. Scenario can initialize parameters and pass them to Engine. Most important, Scenario filters Engine's output. Scenario takes Engine's dull boring simulation output and turns it into appealing, meaningful, usually graphical, output. Engine is thus free to do what it does best: model execution, producing output. Scenario then does what it does best: facilitating visualization of the output as "answers".

Scenario detail is unique to each model. MOOSE has a toolkit of visualization instrumentation for reuse. This kit includes dials and gauges, like those seen in an automobile or those which measure progress when installing software, as well as simple xy plot graphs and terrain maps. This toolkit is extensible and as new models are developed, the toolkit grows and becomes more useful. Nonetheless, some simulation output isn't necessarily amenable to graphical realtime treatment, and there is a very necessary role for traditional methods of analysis.[26–28] MOOSE can support this in two ways: Engine can send output for this purpose to a file separate from that examined by Scenario; alternatively, Scenario can direct some of Engine's to such a file. Either way, further analysis of such output can then be handled by additional software provided by model author, *e.g.*, MATLAB. Such software can be invoked from Scenario, or it may be completely external to MOOSE.

4.7. MOOSE Object Store (MOS)

The MOOSE Object Store (MOS) component of the MOOSE Library holds object data. MOS does for objects much of what MMR does for models. MOS works with Engine and Scenario, in similar fashion to the way MMR works with Modeler and Translator. MOS manages object persistence and distributed objects. An object enters MOS either to hibernate (persistence), or to move to a different host (distributed). An object in a MOOSE component such as Engine decides to go into MOS. MOS accommodates this. The object can re-emerge at any MOOSE location, either on the same host or a different host, but not to two (or more) locations at the same time. What is supported here is *moving not copying*, corresponding to the real-world constraint that there cannot be more than one copy of a given object in existence at any "moment".

Objects are "flattened" as they go into MOS, and "inflate" again when they are come out. Primitive types, such as int, real, and string, are stored as themselves. All other types are abstract data types (ADT's) formed from primitive types and/or other ADT's, and are recursively (eventually) flattened into primitive types. For example, an image in .gif format belongs to a class (ADT) with two attributes: an array of byte and an integer length.

An object can persist by sending itself to the MOS. The object can come back from the "hibernating" to the "active" state with a special form of its constructor that works with MOS. This technique is extensible to support distributed operation by having the two operations occur on different hosts. An object can move by placing itself into MOS with a request to be moved to a specific host; or, an object can place itself into MOS and wait for a *request* which will cause it to move to some location which need not be known when the object was stored. Each MOOSE host has an MOS, and several MOS's collaborate to find and retrieve objects, so that an object can move from any MOOSE host to any other MOOSE host where MOS is active.

Although the architecture permits MOS to be capable of distributed operation, this is not our present focus in MOOSE. We have decided to focus on distributed model definition rather than distributed model execution. Thus, MOS operates in support of model execution on a single host only at this time, so that only persistence (and not distributed objects) is supported. Work on distributed objects to function as described above is currently underway.

5. IMPLEMENTATION DETAIL: SOME IMPORTANT CLASSES; PLATFORMS

Blocks: This section describes some key classes in MOOSE back-end software, comprised of Translator and Engine. First we discuss the *Block* class: MOOSE models are multimodels built mostly of dynamic models, and every dynamic

model has a structure (subordinate elements) and a topology (how those subordinate parts are connected). FSM's, for example, have states connected by arcs labeled with predicates that control state transitions; and, FBM's, for example, have function blocks connected by traces which carry output of one block to input of another. In MOOSE, every subordinate element of every dynamic model (*e.g.*, state of an FSM, functional block of an FBM, is an object of a derived class of the base class *Block*, so called for historical reasons (our first dynamic model type was FBM). This homogeneity facilitates model refinement and so is an underlying support for multimodels of all kinds, most especially heterogeneous multimodels.

Clusters: Associated with each Block object is a structure known as Clusters, which hold pointers to objects known to belong to classes that have certain relations to the object. Each dynamic model method belongs to a class, but the identity and true nature of each block within that dynamic model can be bound as late as every time the method is dispatched. Clusters are searched as needed to identify the block objects to associate with each element of a dynamic model, just before each execution of the dynamic model. This dynamic block binding facilitates dynamic multimodeling. It is also anticipated to support distributed simulation when MOOSE moves onto the web.

Context: MOOSE engine is event-scheduled using ooSim. This is essentially transparent to the model author, who only specifies the time step for each model within each event chain (or one overall time step if all time steps are the same). Event chains can terminate themselves, or they can end when the simulation clock reaches a specified time. The consequence of event scheduling is that all events, such as one execution of the code of a dynamic model are called from the ooSim event Dispatcher. There cannot be any loops in the dynamic models: the equivalent of loop behavior is attained by propagating event chains. Each dynamic model is a method of some class. ooSim does not use global symbols, so to have the equivalent of static local variables *private to each object*, a *Context* structure holds this information. Contexts are generated automatically by Translator for dynamic models. This facilitates event-scheduling.

Glist: MOOSE needed a base class for lists, because MOOSE has lots of lists to manage. The *Glist* class is this base class. A Glist object is a dynamically allocated array of pointers. When insertion is performed, the array senses when it is full, and automatically expands, in a way transparent to the caller. Glist is not a linked list, it is an array, so it has speed and safety advantages relative to linked lists. Derived classes of Glist are made type-safe [29] by appropriate casts in derived class declarations (*not* in calling code!). There are specialized methods that were needed for one derived class or another, and were put into the Glist base class, and so became available for all derived classes, present and future, to use. First Glist was handy in Translator, then it was reused in Engine runtime support; most recently, it has become the foundation of the container classes which facilitate representing aggregation.

Dynamic: The present MOOSE implementation includes several of dynamic models: FSM, FBM, EQN, and RBM. We see needs for other kinds of dynamic models, such as Petri nets, System Dynamics models, Fuzzy models, and perhaps others. There is a requirement that MOOSE be painlessly extensible to new dynamic model types. The *Dynamic* class is an abstract base class[13] in Translator, from which all current dynamic model classes internal to Translator are derived, and which will facilitate creation of new dynamic model types in future.

Portability: Platforms and Languages: We chose two target platforms for MOOSE: the first is Sun Solaris dialect of System V Unix; the second is Microsoft Windows NT. The code also seems to run under Windows95 but we do not develop under Windows95. The Unix platform was chosen for convenience, as it is ubiquitous in our Departmental environment, as it is across academia. The Windows NT platform was chosen because it runs not only on the IBM-compatible PC with Intel x86 CPU, but also on RISC processors like Digital's Alpha AXP, the PowerPC, and MIPS RISC.[30] For MOOSE programming languages, we chose TclTk for our components with GUI's (Modeler and Scenario); and, C++ for our back end components (Translator, TTL, and Engine runtime support). MOOSE back end code has been compiled under MS Visual C++, Borland C++, Borland Turbo C++, and g++.

6. CONCLUSIONS AND PLANS

TclTk is our programming language for Modeler and Scenario GUI's. As TclTk neither enforces nor facilitates object-oriented methodology, we are looking at ways to retain the benefits of TclTk while improving the reusability and extensibility of the code. The MOOSE Model Repository (MMR) has been a substantial step in the right direction, offloading from Modeler the job of keeping track of all the details of a model. We will also explore object-oriented alternatives to TclTk. Scenario has a difficult job: it must facilitate visualization even though every model is different in surface appearance. Scenario is also presently written in TclTk so the remarks above regarding lack of object-oriented support apply as well. Nonetheless, we are working on building a toolkit of popular and reusable dials, gauges, graphs, and clip art, and are looking at XF and SpecTcl GUI builders. Our HCI needs constant review and improvement, especially as new immersive technologies beckon. It is a fundamental tenet of MOOSE and OOPM, that the HCI must fit the model author like a glove. Our objective is to make it *fun* to use the MOOSE HCI. Accordingly we are prepared for the HCI to evolve.

The present MOOSE implementation includes several kinds of dynamic models: FSM, FBM, EQN, and RBM. We see needs for other kinds of dynamic models, such as Petri nets, Rule based models, Fuzzy models, and perhaps others. Aggregation and the implications of aggregation pose interesting questions, especially as we distinguish among containment, usage, composition, and association. Although we have significant results, more work lies ahead. We plan to take MOOSE onto the worldwide web, to distribute model execution. For web-based operation, a plan is underway to embed distributed model execution within a MOOSE method using CGI (Common Gateway Interface). We also plan to evaluate a Java alternative in this context.

REFERENCES

1. P. A. Fishwick, "Extending object oriented design for physical modeling," *ACM Transactions on Modeling and Computer Simulation* , July 1996. Submitted for review.
2. I. Sommerville, *Software Engineering, Fourth Ed.*, Addison-Wesley, 1992.
3. P. A. Fishwick, "The role of process abstraction in simulation," *IEEE Transactions on Systems, Man and Cybernetics* **18**, pp. 18 – 39, January/February 1988.
4. P. A. Fishwick, "Abstraction level traversal in hierarchical modeling," in *Modelling and Simulation Methodology: Knowledge Systems Paradigms*, B. P. Zeigler, M. Elzas, and T. Oren, eds., pp. 393 – 429, Elsevier North Holland, 1989.
5. V. Berzins, M. Gray, and D. Naumann, "Abstraction-based software development," *Communications of the ACM* **29**(5), pp. 402 – 415, 1986.
6. P. A. Fishwick and K. Lee, "Two methods for exploiting abstraction in systems," *AI, Simulation and Planning in High Autonomous Systems* , pp. 257–264, 1996.
7. P. A. Fishwick, *Simulation Model Design and Execution : Building Digital Worlds*, Prentice Hall, 1995.
8. K. Lee and P. A. Fishwick, "Semi-automated method for dynamic model abstraction," in *SPIE Conference Proceedings*, 1997.
9. B. P. Zeigler, *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, 1990.
10. G. Booch, *Object-Oriented Analysis and Design with applications, 2nd ed.*, Addison-Wesley, 1994.

11. A. J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.
12. B. P. Zeigler, *Objects and Systems*, Springer-Verlag, 1997.
13. B. Stroustrup, *The C++ Programming Language, second edition*, Addison-Wesley, 1991.
14. G. Booch and J. Rumbaugh, *Unified Method for Object-Oriented Development*, Rational Software, 1995.
15. M. Gardner, "Mathematical games," *Scientific American, Oct 1970* , 1970.
16. M. Gardner, "Mathematical games," *Scientific American, Feb 1971* , 1971.
17. M. Gardner, *Wheels, Life, and Other Mathematical Games*, publisher, 1983.
18. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990. Second Edition.
19. P. A. Fishwick, "A visual object-oriented multimodeling design approach for physical modeling," *submitted April 1997 to ACM Transactions on Modeling and Computer Simulation* , 1997.
20. P. A. Fishwick and B. P. Zeigler, "A Multimodel Methodology for Qualitative Model Engineering," *ACM Transactions on Modeling and Computer Simulation* 2(1), pp. 52–81, 1992.
21. P. A. Fishwick, "A Simulation Environment for Multimodeling," *Discrete Event Dynamic Systems: Theory and Applications* 3, pp. 151–171, 1993.
22. R. Orfali, D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley and Sons, 1996.
23. R. M. Cubert, "The oosim object oriented simulation library," Tech. Rep. TR951230, University of Florida CISE Simulation Group, 1995.
24. P. A. Fishwick, "Simpack: Getting started with simulation programming in c and c++," in *Winter Simulation Conference WSC'92 Proceedings*, J. J. S. et al., ed., pp. 154–162, 1992.
25. M. H. MacDougall, *Simulating Computer Systems : Techniques and Tools*, MIT Press, 1992.
26. J. A. Payne, *Introduction to Simulation : programming techniques and methods of analysis*, McGraw-Hill, 1982.
27. A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, 1991.
28. G. S. Fishman, *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley & Sons, 1973.
29. B. International, *The World of C++*, Borland International, Scotts Valley, CA, 1991.
30. K. Siyan, *Windows NT Server*, New Riders, 1995.

AUTHORS' BIOGRAPHIES

Robert M. Cubert   is a PhD student in CISE at University of Florida. His research interest is object-oriented distributed simulation. He holds BS degrees in EE from MIT and in Zoology from University of Oklahoma, and an MS in Computer Science from University of Oklahoma. He spent 3 years on Computer Science faculty at California State University, Sacramento, and has a decade of industry experience writing software for realtime control systems and data communications.

Paul A. Fishwick   is an associate professor in the Department of Computer and Information Sciences at the University of Florida. He received the BS in Mathematics from the Pennsylvania State University, MS in Applied Science from the College of William and Mary, and PhD in Computer and Information Science from the University of Pennsylvania in 1986. He also has six years of industrial/government production and research experience working at Newport News Shipbuilding and Dry Dock Co. (doing CAD/CAM parts definition research) and at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a senior member of the IEEE and the Society for Computer Simulation. He is also a member of the IEEE Society for Systems, Man and Cybernetics, ACM and AAAI. Dr. Fishwick founded the comp.simulation Internet news group (Simulation Digest) in 1987, which now serves over 15,000 subscribers. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990) and he is on the editorial boards of several journals including the ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics, The Transactions of the Society for Computer Simulation, International Journal of Computer Simulation, and the Journal of Systems Engineering.

MOOSE Model Repository
in Distributed Modeling *

Robert M. Cubert
and
Paul A. Fishwick
Department of Computer & Information Science and Engineering
CSE Room 301
University of Florida
Gainesville, FL 32611-6120

Abstract

Multimodeling Object-Oriented Simulation Environment (MOOSE) is an implementation of Object Oriented Physical Modeling (OOPM) methodology. MOOSE components include a Model Repository (MMR), a Conceptual Modeler GUI, several Dynamic Multimodel Editors, a Translator which takes model definitions to simulation programs, support for simulation output visualization using VRML (Virtual Reality Markup Language) in a web-based setting. A Distributed Modeling Markup Language (DMML) has been developed for communicating among MOOSE components, and between the WorldWide Web and MOOSE. DMML has similarities to the Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL), and to the Department of Defense DMSO (Defense Modeling and Simulation Office) HLA (High Level Architecture) OMT DIF (Object Model Template Data Interchange Format). MMR uses DMML to support distributed modeling, allowing geographically dispersed model authors to form virtual workgroups, and facilitating model reuse. MOOSE components are accessible in several compatible configurations which support different needs, including the primary configuation which uses a web browser plug-in.

---

*email rmc@cise.ufl.edu; fishwick@cise.ufl.edu

1 Introduction

MOOSE (Multimodeling Object-Oriented Simulation Environment) is a software environment combining mostly-visual metaphors with refinement of behavioral abstractions using a wide range of dynamic multimodels to facilitate object-oriented analysis and design in support of developing simulations. MOOSE can be configured as a general-purpose tool to facilitate modeling, or, with the addition of specialized class libraries, can serve as an application framework in a vertical niche, such as for modeling ecosystems. MOOSE is based on the OOPM (Object-Oriented Physical Modeling) approach developed by the second author, and thus bears similarities to familiar OOA&D (Object-Oriented Analysis and Design) techniques such as UML (Unified Modeling Language).[1] The MOOSE Conceptual Model represents classes, relations among classes such as composition and specialization, objects, and relations among objects. MOOSE dynamic multimodels represent a powerful and formal approach to behavioral refinement which provide mostly-graphical metaphors for several alternative representations to better suit the varying taste and background of diverse groups of model authors, including such representaions as finite state machines, rule-based models, and others. MOOSE uses the power of VRML (Virtual Reality Markup Language) to express geometry; thus, in MOOSE, output visualization as well as static exploration of a model's geometry are leveraged with VRML. Additional leverage is provided by the WorldWide Web: MOOSE is being packaged as web-based software to facilitate broad painless distribution not only of MOOSE software itself, but also of the models developed using MOOSE, and to minimize platform-dependence. MOOSE supports distributed model definitions, model reuse, and collaborative model development. At simulation run-time, MOOSE supports a federation of simulations as contemplated by the Department of Defense DMSO (Defense Modeling and Simulation Office) HLA (High Level Architecture).

2 A MOOSE Overview

MOOSE provides a mostly graphical way for model authors to express, define, communicate, and think about models. Once entered, model definition information becomes persistent: it is retained by the system and is available for later use. One such use enables me to resume work on a model next week where I left off today; a more interesting use is to send a model definition to the MOOSE Translator, whose output is a machine-generated simulation program in the C++ programming language, unambiguously derived from the model definition. When a machine-generated simulation source program is translated to binary and augmented with runtime library support, a simulation executable we call a MOOSE Engine is created, which can run once or many times as the simulation corresponding to

---

[1]Caveat: comparing MOOSE with UML is not quite appropriate; as MOOSE is an implementation of OOPM, it would be more appropriate to compare MOOSE with a tool that implements UML, which is outside the scope of this paper

the model. The Engine is available not only to the model author, but to other users as well; it can be coupled to an output visualizer using TclTk or VRML.

In the original development of MOOSE, model definition persistence was accomplished via textual format, in a set of flat ASCII files comprising a model definition. This approach had (and still has) a number of benefits, including: such model definitions are compact, relatively easy to read, understand, and even modify if need be; model definition files get backed up as part of local system backups; models can be put on diskette, into a .zip archive, or ftp'd; it also is a software engineering tool to eliminate development bottlenecks. This incarnation of MOOSE is standalone software, with no provision for sharing, thus limiting reuse; moreover, this MOOSE can be used on a machine only after MOOSE software is obtained and installed on that machine.

Subsequent progress on MOOSE has continued on several fronts, among which two are the focus of this paper: (1) a new approach to model definition persistence we call "model repository"; and (2) making the modeling environment web-based (as in World-Wide Web).

The MOOSE Model Repository (MMR) communicates using the connection-based TCP (Transmission Control Protocol) over IP (Internet Protocol) with producers and consumers of model definitions, as an alternative to the local-file-based model definitions mentioned above. Model authors still interact with the mostly graphical interface, but persistent model definitions reside within MMR; similarly, MOOSE Translator still converts model definitions to C++ simulation programs, but model definitions are from MMR rather than local files. These changes are essentially transparent to model authors. Benefits include: (1) a model defined on machine A can be translated on machine B; (2) a model can be defined and/or translated on a machine with no (or limited) local persistent store; (3) models reside where they can best be catalogued, indexed, browsed, backed up, and otherwise maintained, without distracting model authors from their primary focus. MMR also permits models to be shared in a way that was not available before: for example, a model defined on machine A can be referenced on machines B and C, so Ann's model is available to Ben and Cal. This not only (4) increases reuse potential; it also (5) provides an environment to support collaborative development, a distinct benefit. Disadvantages include reliance on network connections and consumption of network bandwidth. MMR's may from time to time start and stop, so the MOOSE universe may have any number of MMR's. MMR's know about one another, can forward requests to their peers, and can share model information. But MMR does not, in and of itself, make MOOSE "web-based".

Fortunately, the worldwide web *does* offer a way for MOOSE to be a web-based modeling environment. Our "litmus test" for whether software is "web-based" is: (1) that it require no installation of separate software, and (2) that it rely on communication conventions of the web (eg, URL's). There are several ways to meet these criteria, combining some or all of the following: HTML (HyperText Markup Language) JavaScript, Dynamic HTML, Java applets, and browser plug-ins with or without LiveConnect[2] The primary MOOSE configu-

---

[2]When a plug-in is requested for the first time, a request pops up asking the user whether it's ok to

ration is web-based: a browser plug-in LiveConncet'ed with Java applets, based on HTML, with a sprinkle of JavaScript. MMR can be thought of as a server-side phenomenon, so the standard web-based configuration does not include an MMR on the client. We contemplate supporting three additional configurations, one web-based and the other two not web-based. These are (2) a web-based runtime-only (Engine and visualization) configuration; (3) a "power-user" configuration providing a local MMR and/or a local Java-based GUI and/or a TclTk-based GUI, which operate out of the same consistent plug-in top level as the web-based configurations, and which may be more appropriate where network bandwidth and/or security issues are paramount. Finally there is (4) the original standalone configuration of MOOSE. The last two configurations are not web-based because they require MOOSE and possibly TclTk to be installed on the local machine.

3   MOOSE Models

Above we alluded to MOOSE model definitions in a cursory way. Here we provide sufficient detail to motivate and support an explanation of DMML (distributed model markup language), the model definition language used in MOOSE. Although a model author may not see DMML, the components of MOOSE with which a model author interacts use DMML to communicate with each other; in particular, DMML is the language in which GUI components communicate with MMR.

A MOOSE model of a physical system is expressed in terms of a conceptual model, a dynamic model, and a geometry model. DMML has a representation for each of these; additionally, DMML has a representation to deal with establishing a web-based environment in which to work. Before getting to these parts of DMML, we first elaborate the corresponding parts of MOOSE. The treatment here is necessarily brief; the interested reader is referred to our earlier work (ref: see Fishwick ref 8, also our spie 97 paper?)

3.1   Model Interfaces

A model interface is an expression of that model's public face to the world at the highest level of abstraction, the contract it agrees to follow, while encapsulating (hiding) details of its behavior and state. Although not required for an effort focused solely on modeling, the model interface is useful when time comes for the simulation which corresponds to model A to interact with the simulation coresponding to model B. This is a topic of great interest for example to the Department of Defense DMSO (Defense Modeling and Simulation Office) HLA (High Level Simulation Architecture) as it contemplates a federation of

---

automatically download, install, and start the plug-in. All of this happens automatically and doesn't require the browser to be shut down and restarted; accordingly, we consider this sufficiently transparent as not to constitute fetching and installing software. Similarly, a Java applet must be downloaded, but the browser does this for us, and we do not consider this to be fetching and installing software.

independently-developed simulations being able to communicate. We want MOOSE to be able to participate in this area, so DMML defines model interfaces.

3.2 Conceptual Model

A conceptual model expresses abstractions of relevant aspects of the application domain (ref: Coplien p.201), in the form of classes, instances, relations among classes, and relations among instances (ref: Hill p.104). A conceptual model often has multiple views (ref: Booch p.172) to express various kinds of relations, notably composition and inheritance hierarchies, as well as possibly to highlight key use-cases (ref: booch p.158 −¿ Jacobson ref @ booch p.501). The conceptual model embodies Hill's "static aspect" (ref: Hill p.104), and is similar to, although simpler than, the diagrammatic representations used in UML (Unified Modeling Language) (ref: see Fishwick's paper ref 3..5).

3.3 Dynamic Model

Behaviors of a physical system are represented by its Dynamic Model, comprised of a set of dynamic multimodels, each expressing a virtual method of a class defined in the conceptual model. Multimodeling selectively refines behaviors as required to capture an appropriate degree of fidelity in the most natural metaphor(s). So as not to impose a particular single approach on model authors with diverse problems, backgrounds, and preferences, MOOSE offers an eclectic mix of popular representations, including finite state machines, functional block models, equation models, rule-based models, and system dynamics models; each supported by a GUI. Dynamic multimodel types can be arbitrarily mixed and matched by a model author; for example, one state of a finite state machine may consist of a functional block model with three blocks, one of which is a rule-based model, another a finite state machine, and the third an equation model. A dynamic multimodel may extend to arbitrary depth, and its elements may be of any dynamic multimodel type. Internally, a MOOSE dynamic multimodel is a collection of individual units, and is itself a unit which can be included in a larger collection (ref: Gamma's "Composite" pattern, p.163). It is important to distinguish a dynamic multimodel's method signature from its method definition: the former belongs to the Conceptual Model; the latter to the Dynamic Model.

3.4 Geometry Model

Some parts of a physical system being modeled can be abstracted away; other parts have geometric properties that are important to capture so that the model will have requisite fidelity. Similar to behavioral abstraction, geometry is subject to composition and inheritance following the structure provided by the conceptual model (especially by its relations). The geometry primitives are attributes of classes of the conceptual model. We originally

experimented with TclTk as a language in which to define geometry attributes, including having the MOOSE Translator emit TclTk "glue" code to help with output visualization at simulation runtime; our current efforts define geometry attributes in VRML (Virtual Reality Markup Language); or, more precisely, permit the model author to do so, with resultant web-based runtime output visualization, involving TCP/IP communication between MOOSE Engine and a Java applet which in turn interacts, via shared memory, with the CosmoPlayer 2.0 browser plugin. Determining ways to best effect composition of geometry elements is an area of active research.

4   A sample model and use-cases

Partly because we live within the Florida ecosystem, and partly because of the wider importance of fragile, irreplaceable natural resources like the Everglades, we have chosen an Everglades ecological sample model. One of the waterbirds dependent on the freshwater marsh habitat of the Everglades is *Rostrahamus sociabilis*, the snail kite, an endangered species whose range has shrunk to just a few percent of what it once was, partly as a result of ill-advised flood-control projects (ref: Myers, p.354). Because of its endangered status, the snail kite is the subject of simulation studies aimed at understanding what affects this species, in order to find ways to improve its lot.[3] The snail kite's food source is the apple snail, a species whose success in large measure depends on temperature and hydrology. Thus to construct an ecological model for snail kites, it may well be necessary to have an ecological model of the apple snail. We might for example hypothesize that diverting water reduces snail population, which in turn reduces the ability of the snail kite to eat and to breed, and set out to build a model so that we can run the corresponding simulation to test this hypothesis.

Consider this undertaking. The model author, about to begin developing a MOOSE snail kite model, is an expert on birds but knows little about snails. Perhaps by word of mouth from a colleague, or through a websearch using a search engine (Excite, Yahoo!, Lycos, etc.), or perhaps by querying MMR using search techniques we envision but have not yet developed, the snail kite model author ascertains that a snail expert[4] has developed an apple snail MOOSE model. What are some use-cases we can anticipate?

The kite model author views the snail model, to get a sense of what factors the snail model author thinks are important, and learns that these are temperature and hydrology. This involves an MMR search, followed by use of the MOOSE conceptual modeler and dynamic multimodel editor GUI's to display the snail model. Application domain knowledge is thus transferred via a MOOSE model.

---

[3]Such efforts are part of a study of the Everglades ecosystem being undertaken by the U.S. Department of the Interior's ATLSS project.

[4]Phil Darby et al. at University of Florida

The kite model author now faces a choice: to incorporate the snail model into the kite model at model definition time, or to have the kite simulation "federate" with (the output of) the snail simulation at simulation runtime.

Incorporating the snail model into the kite model again involves a choice: the snail model definition can either be referenced or it can be copied. A reference will allow improvements in the snail model to auomatically appear in the kite model; a copy ensures that later changes to the snail model won't make the snail model unusable in the kite model. Each approach has merits and risks. The choice depends on the relationship between the model authors. A heuristic: if the authors belong to a collaborative workgroup, use reference; if they have no ongoing relationship, use copy.[5]

A simulation federation also involves choices, of which we set forth two here: (1) the kite simulation can simply run the snail simulation and use snail simulation output, as something like a table of snail populations to "feed" the kite simulation; (2) the kite simulation can run the snail simulation synchronously, controlling its rate of progress, in effect "latching" the snail simulation output, and calling snail simulation public methods from the snail model interface published through the MMR, to obtain snail data.

5 Relation of DMML to other Developments

CORBA (Common Object Request Broker Architecture) is the work of the Object Management Group (OMG), an industry consortium whose stated objectives include the adoption of standards for managing distributed objects. OMG developed an Object Management Architecture (OMA) which includes CORBA as the "bus" which forms the basis for managing distributed objects. CORBA relies on all specifications being provided in IDL (Interface Definition Language) and anything that attaches to the CORBA bus has to be defined in terms of CORBA IDL. This permits objects and services written in different languages on different platforms to work together.

Microsoft COM (Component Object Model) is a technology that one uses to define components, which are objects that encapsulate both data and code, and provide a well-specified set of publicly available services. DCOM (Distributed COM) extends COM to remote operations. Clients have access to a COM object only through its interfaces, defined through the COM IDL (Interface Definition Language). Features we find attractive to MOOSE include stateless poolable objects (ref: sessions p.466), and the use of Monikers as names that uniquely identify COM objects, associating a name with its referent, putting it into its running state if it isn't already, and returning an interface pointer to it (ref: MS Visual C++ 5.0 online documentation).

The Department of Defense through its DMSO (Defense Modeling and Simulation Office)

---

[5]As often happens, we gain insight into the answer to a question in an abstract domain by posing and answering a corresponding question in the underlying physical domain.

has created its HLA (High Level Architecture) for modeling and simulation. HLA has OMT (an object model template) which in turn has a DIF (Data Interchange Format), intended to allow simulations to federate by knowing something about the public interfaces exposed by each other's corresponding models. (Remember that models don't federate, simulations do.) Features we find attractive to MOOSE include (1) the idea of a federation of simulations, which interact via knowledge of one another's public interfaces, and having MOOSE able to operate within an HLA-compliant federation. We also consider (2) MMR to be a good fit with the HLA MSRR (Modeling and Simulation Resource Repository (ref: http://www.msrr.dmso.mil). Syntax of the DIF has much in common with that of DMML. A practical impact on DMML is that any simulation which implements a model has the opportunity to register with the MMR holding the model definition; thus, prospective users of a simulation can first examine the model, and, after confirming its appropriateness, select any simulation which is an implementation of that model. This provides a mechanism not only for multimodeling but also for competitive operations, patterned after the "yellow pages".

6   The DMX Control Panel

DMML is not only the language spoken between MOOSE GUI's and the MMR to express model definitions; it is also the language spoken to the DMX control panel browser plug-in. In the latter role, DMML can start a local MMR or connect to a remote MMR; it can allow the user full access to its capabilities, provide a reduced set of operations, or put on a completely "canned" show. To understand how this works, one must know a little about web browser plug-ins and MIME types.

A web browser such as Netscape Communicator Professional Edition version 4.04 can be extended with platform-specific programs known as plug-ins. In an MS Windows NT/95 environment, for example, a plug-in is a .DLL which becomes part of the browser process, sharing its address space. When the browser is activated it looks in certain directory for plug-in .DLL files. A plug-in .DLL is created according to certain rule with certain resources that, for example, associate it with a MIME type (see below)) and a file extension. When the browser sees such a plug-in .DLL, it ascertains its MIME type and file extension. Then, should the browser encounter a URL (Universal Record Locator) or an HTML "¡EMBED¿" tag "SRC" attribute whose file extension matches that of the plug-in, an instance of the plug-in will be activated with the specified URL or SRC attribute as its input.

MIME (Multipurpose Internet Mail Extensions) types (ref: http://www.oac.uci.edu/indiv/ehood/MIME/1 are registered with IANA (Internet Assigned Numbers Authority). The set of documents referred to below are "Requests for Comment" dated November 1996 and obsolete the earlier RFC's 1521, 1522, and 1590 (March 1994) on the same subject.

- RFC 2045: MIME Part One: Format of Internet Message Bodies

- RFC 2046: MIME Part Two: Media Types

- RFC 2047: MIME Part Three: Message Header Extensions for Non-ASCII Text

- RFC 2048: MIME Part Four: Registration Procedures

- RFC 2049: MIME Part Five: Conformance Criteria and Examples

The MIME type "plugin/x-dmx" and its associated file extension ".dmx" are associated with DMML, and the DMX Control Panel browser plug-in. Thus, when Netscape web browser sees an HTML document containing, for example, the HTML "¡EMBED¿" tag

```
<EMBED SRC=http://www.cise.ufl.edu/~rmc/sim/test.dmx HEIGHT=450 WIDTH=750>
```

the result is to activate an instance of the plug-in, in this case the DMX Control Panel, and to pass the SRC URL test.dmx to it as an input stream. The SRC URL directs the behavior of the control panel: it can select modes of operation, enable or disable features, even download a specific model, put on a complete model building demo, or perform a simulation run with output visualization. Normally we do not rely much on these capabilities; instead, we enable everything and allow the user to interact with the DMX control panel's full capabilities.

The DMX Control Panel interacts with the MMR, and with GUI's such as the MOOSE Conceptual Modeler and Dynamic Multimodel Editors. Any or all of the following techniques may be in use, depending on the configuration: (1) TCP/IP is used to communicate with MMR. (2) The plug-in spawns processes when a local installationof MOOSE is present, such as the TclTk-based GUI. (2) When the Java applet versions of the GUI's are present, in the web-based configuration, MOOSE uses LiveConnect and the JRI (Java Runtime Interface) to allow the plug-in to activate instanaces of Java applets and to call their public methods.

The DMX Control Panel thus provides a compatible basis for all supported configurations. It has three interfaces: (1) a dialog-based GUI using Microsoft Visual C++ MFC (Microsoft Foundation Classes) (2) a URL interface, either through the URL value of the "SRC" attribute of an "¡EMBED¿" HTML tag, or by direct invocation of the browser on a URL of the plug-in's file extension (".dmx"), (3) an API (through JRI) available to Java applets and thus indirectly to JavaScript as well.

7 DMML Example

The DMML example below is a heavily edited portion of the apple snail model presented above. The first statement is an example of the ability of DMML to control web-based operations, as mentioned above. In this case we just identify the (remote) MMR and

indicate that it is active. Next we see a model interface for the AppleSnail model. This is the public face which the model shows to the world. We are able to see that the model outputs an 8 by 12 grid of snail population values with a time scale of months, a time step of 0.1 month, over a time interval of 36 months.

Then comes a conceptual model for AppleSnail. We see some classes defined, some composition (has_a) and specialization (is_a) relations, and examples of geometry with inline VRML and an inline dynamic multimodel, in this case a rule-based model. It should be emphasized that inline definitions are only one approach, and that a conceptual model can instead reference its dynamic and geometry components rather than include them.

```
Repository use pegasus is active

Model Interface AppleSnail
   exports snailpop real array 8 by 12
   time step 0.1 month
   time limit 36 month

Conceptual Model

Class Marsh
   has 8*12 CellS
   geometry vrml at c1g1.wrl

Class CellS is_a Container of Cell

Class Cell
   has_a SnailPopulation
   has_a Hydrology
   has_a Temperature

Class SnailPopulation
   has_a EggPopulation
   has_a JuvenilePopulation
   has_a AdultPopulation
   behavior HatchEggs is rule_based_model =
     rule_based_model HatchEggs
        input real hatch_rate
        returns int hatched_eggs
        block1
           premise      month >= 3 AND month < 10
           consequence Marsh::M8
```

```
    block2
       premise      month >= 10
       consequence SnailPop::M6
    topology block1 block2
       IN1 goes to B1 IN1
       OUT1 comes from B1 OUT1
       OUT2 comes from B1 OUT2
       Attribute month goes to B2 IN2
       IN1 goes to B2 IN1
       OUT1 comes from B2 OUT1
       OUT2 comes from B2 OUT2
  behavior PopEvolve is system_dynamics_model at c4m2.tpf
  geometry vrml at c4g1.wrl
  real maturation_time
  real juvenile_growth_rate
  real preadult_death_rate

Class AdultPopulation
  is_a SnailPopulation
  geometry vrml =
    Transform
      translation 2 0 0
      children Shape
        appearance Appearance
          material Material { diffuseColor 0 0 1.0 }
        geometry Sphere { radius 0.5 }

Dynamic Model
```

8   section

OOPM: An Object-Oriented Multimodeling and Simulation Application Framework

Robert M. Cubert and **Paul A. Fishwick**
Department of Computer and Information Science and Engineering
CSE Room 301
University of Florida
Gainesville, FL 32611-6120

OOPM, or Object-Oriented Physical Multi-modeling, is an application framework for modeling and simulation under development at the University of Florida. It extends object-oriented program design with visualization and a definition of system modeling that reinforces the relationship of model to program. Model authors interact with OOPM via graphical user interface, which captures model design, translates models to simulation programs, controls model execution and provides output visualization. Distributed Model Repository facilitates collaborative and distributed model definitions and model reuse. Translator converts model definition to a simulation program in C++, then compiles and links this simulation program, adding run-time support and creating an executable which runs under control of Scenario to provide output visualization using Virtual Reality Modeling Language. A variety of model types may be freely combined through heterogeneous multimodeling, which is the basis for geometry and dynamic behavior models.

Keywords: Multimodel, object-oriented modeling, OOS, model abstraction, object-oriented physical modeling, visualization, application framework, OOPM

1. Introduction

OOPM, ("Object-Oriented Physical Multimodeling") is an application framework providing components and patterns for modeling and simulation [1] being developed at the University of Florida. It embodies an approach to modeling and simulation which not only tightly couples a model author into the evolving modeling and simulation process through an intuitive human-computer interface (HCI), but also helps the model author with any or all of the following: (1) to think clearly about, to better understand or to elucidate a model; (2) to participate in a collaborative modeling effort; (3) to repeatedly refine a model as required to achieve adequate fidelity at minimal development cost; (4) to build integrated models using existing proven small models as subsystems; (5) to start from a conceptual model which is intuitively clear to domain experts, and to unambiguously and automatically convert this to a simulation program; (6) to create or change a simulation program without being a programmer; (7) to perform simulation model execution and present simulation results in a meaningful way, which facilitate the other objectives above.

In some cases modeling alone, without executing a simulation program, suffices to achieve the model author's objectives, which may be to learn about or better understand a phenomenon or system, or to

41
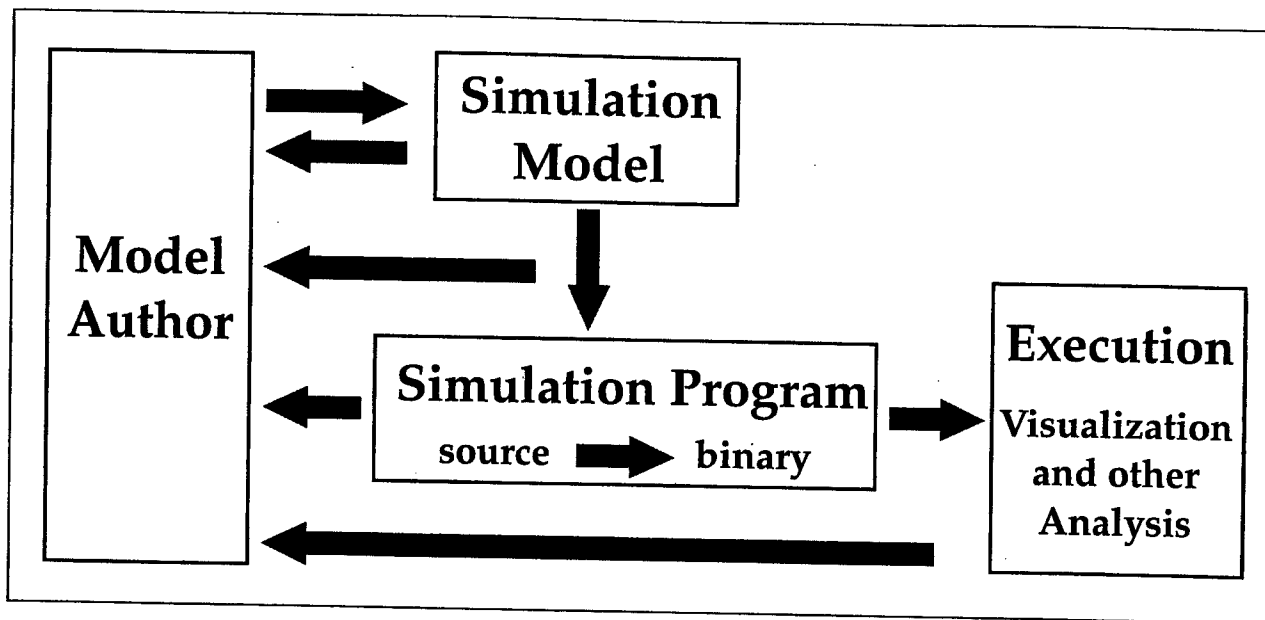
Figure 1. Metamodel for modeling and simulation

communicate with colleagues. Usually, however, a model author wishes not only to model, but also to construct and execute a simulation program (1) to empirically validate the model based on observed behavior; (2) to select or adjust various parameters and values and observe their effects; (3) to measure performance; or (4) to gauge model fidelity and assess its adequacy.

Figure 1 is a metamodel for modeling and simulation, highlighting distinctions among modeling, constructing simulation programs and executing simulations. Model authors collaborate to develop a model, from which a simulation program is constructed, translated to executable form, and executed; the results are visualized or otherwise analyzed. Simulation programs developed directly, without a platform-independent specification (model), are problematic: vulnerable to platform or language change, with low readability, low extensibility, low maintainability and low reuse potential. Even when modeling is used, problems remain: expensive duplication of effort, low quality of reproduced "external" subsystems, limitations in expressivity of modeling frameworks, and limitations on the feasibility envelope imposed by complexity of the development process.

Currently the Modeling and Simulation (M&S) community suffers all these problems. Efficiency and productivity of model authors is low, evidence being that work usually cannot be reused or readily integrated into larger new systems [2]. These problems propagate to every realm where modeling and simulation are used. When a model author sketches a "whiteboard model" with annotations, and uses this to describe to programmers the design of a

simulation program to be written, there are pitfalls. The programmers write a program, *but there is not necessarily a relation between the model described and the program produced*. More formal approaches, such as requirements specifications and a traceability matrix, reduce ambiguity but introduce an unmanageably complex representation and a textual tabular format that is decidedly non-intuitive.

With OOPM, the model author uses visual metaphors in a framework for constructing the model, and then a simulation program is unambiguously and automatically built from the model. Advantages include: (1) built-in model validation [3]; (2) partial automation of the development process; (3) built-in extensibility and flexibility for accommodating unexpected change [4]; and (4) reduction in development time. An additional benefit is the ability to model source systems of greater inherent complexity by integrating "tried-and-true" existing models as subsystems, because, as Booch states, "A complex system that works is invariably found to have evolved from a simpler system that worked... A complex system designed from scratch never works and cannot be patched up to make it work." [4]

Extent of detail in a model reflects the model author's abstraction perspective [5]. Model refinement can produce greater fidelity if required by the model author's abstraction perspective [6] or by external criteria [7]. *Multimodeling* is a recent development [8, 9] which provides multiple levels of abstraction [10] to represent geometry and dynamic behavior of a model. Multimodeling facilitates: (1) model development, selective refinement to achieve required fidelity or model extensibility, and accommodating unanticipated change; (2) integration and

42

reuse of object-oriented distributed simulation models; and (3) extensibility of the framework itself to accommodate future model types.

Figure 2 shows elements of OOPM. The model author interacts with the visual Human/Computer Interface (HCI). The HCI has two graphical user interfaces (GUIs), each supporting a different purpose: "Modeler," which is the Model Author Interface (MAI), and "Scenario," a simulation runtime visualization enabler. The model author interacts with Modeler to design the model. Modeler relies on Distributed Model Repository (DMR, discussed below) for model definitions. Scenario activates and initializes simulation execution, which we name "Engine." Scenario maintains synchronous interaction with Engine, displaying Engine output in a form meaningful to the user, optionally allowing the user to interact with Engine, including modifying simulation parameters and changing the rate of simulation progress. There are two kinds of distributed modeling and simulation: first, distributed model definitions, with various model components defined on different hosts; second, distributed execution running simultaneously on a number of hosts. OOPM focuses on the first, as the area of distributed behavioral multimodels is our primary research interest; hence Scenario, although important, has not received our primary focus.

The OOPM Library consists of: *Distributed Model Repository* (DMR) and *Mobile Object Store* (MOS). MOS holds object data and DMR holds meta-data. DMR stores model definitions defined and used by Modeler, and also used by Translator. Models and model components are available for browsing and

reuse. Class libraries, such as sets for modeling collections and popular geometries for spatial models, are available to the model author. Model definitions can be distributed over several locations. DMR hides location-dependent details. This facilitates collaboration and distributed modeling. Reuse of models, classes and objects is thus mediated by DMR. Reuse examples appear later in this paper.

DMR supports the modeling application framework with more than just a class library: classes are related in such a way that a class is not used in isolation, but within a design encouraged and supported by the framework. DMR stores not only a collection of classes available for reuse, but also relations among models, classes and objects, and classes for geometry and behavior. The language-neutral model definition by which Modeler and Translator communicate with DMR uses the Distributed Multimodeling Language (DMML), developed by the authors and described elsewhere [11].

MOS does for objects much of what DMR does for models. MOS works with Engine and Scenario in a fashion similar to the way DMR works with Modeler and Translator. MOS manages object persistence. MOS is architecturally important; however, as our focus is on modeling, and most MOS issues relate to simulation runtime, our MOS implementation is to date minimal.

Translator is the arrow in Figure 1 between the simulation model and the simulation program. Translator gets from DMR a language-neutral model definition produced by Modeler, and maps it to a computer program for the simulation corresponding to the model, in a Translator Target Language (TTL).
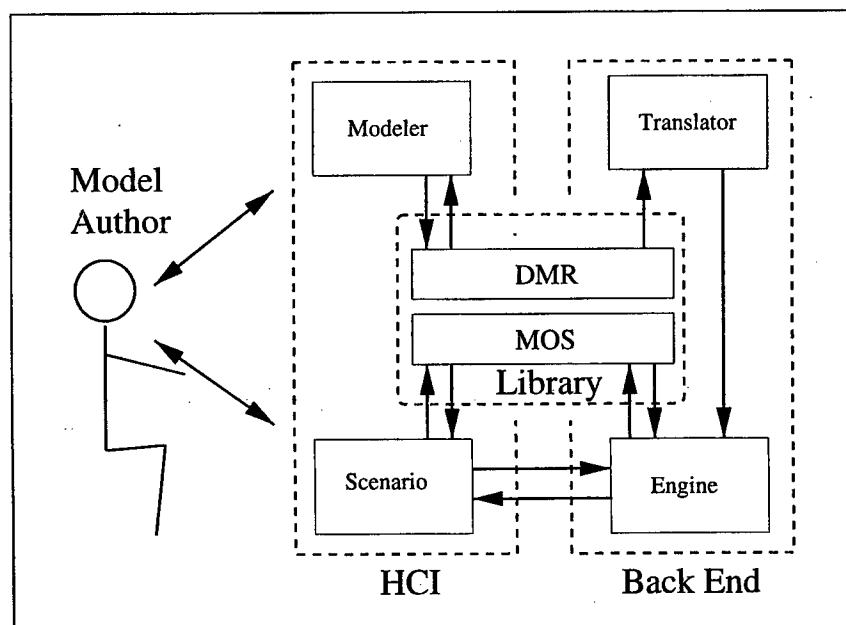
Figure 2. Elements of OOPM; principal interactions indicated by arrows

43

TTL is presently C++; potentially, TTL can be any language. The C++ simulation program emitted by Translator is called *Engine*. Once compiled and linked with Engine runtime support, the Engine executable program is activated under control of Scenario.

To accompany the explanations that follow, a landscape ecology model is used as the example throughout this paper. The model is set in Florida's Everglades and is concerned with population models of apple snails within a two-dimensional spatial array. Apple snails are an important food of the snail kite, a bird which is an endangered species. Reuse of the apple snail model in a snail kite model is an objective of OOPM. The apple snail model may assist in developing the snail kite model. The snail kite model may help scientists learn how to prevent extinction of the snail kite.

The balance of this paper is organized as follows: Section 2 discusses some related work. Section 3 explains the object-oriented approach used by OOPM, and Section 4 explains how and why OOPM employs multimodeling. Section 5 describes visual elements of OOPM, such as conceptual models and dynamic behavior models, and Section 6 describes non-visual elements of OOPM ,such as Translator and DMR. Section 7 states our plans and conclusions.

2. Related Work

Research efforts, software engineering tools, and object-oriented commercial M&S products abound. OOPM differs from each of those, which have been surveyed, in one or more of the following ways: (1) our primary focus is on architecture and representation for distributed model reuse; (2) our model author interface (MAI) is wholly graphical; and (3) multiple-level behavioral abstractions may be represented in any of several alternative ways, each with a formal basis in the literature, each with a community of advocates.

In the DEVS system [12, 13] there are hierarchical behavior models based on proven formalism [14], and a one-to-one relation between model-specification formalism and simulator functionality. Behavior is modeled with one kind of dynamic model based on state machines. OOPM, in contrast, provides five kinds of dynamic multimodels that may be arbitrarily mixed and matched recursively (heterogeneous multimodeling). We are communicating with the DEVS team to determine whether we can combine the modeling strength of OOPM with the strength of the DEVS formalisms and their support for a simulation runtime infrastructure.

Unified Modeling Language (UML) is an object-oriented analysis and design (OOA&D) technique which derives principally from two antecedents: the Booch method [4] and Object Modeling Technique (OMT) proposed by Rumbaugh [15]. A useful survey of these and other object-oriented analysis and design techniques as they apply to M&S is given by Hill [16, 17]. Ways in which UML differs from the OOPM visual modeling environment include: (1) UML dynamics are through one type of dynamic model—state charts—whereas OOPM provides five kinds of dynamic multimodels that may be arbitrarily mixed and matched recursively (heterogeneous multimodeling); and (2) UML state charts are associated with a whole class, whereas an OOPM dynamic multimodel is associated with each method of a class.

OOPM is targeted at making distributed model reuse practical in a visual setting. Java Beans [18], which provides reusable components in a visual builder tool, shares many of our objectives: it has a visual interface, components may be brought from anywhere, and components are self-identifying and self-configuring. The Java Beans GUI "Bean Box" bean builder tool has three areas: a toolbox, a property sheet and a design area where applications are built by associating events of one bean with methods of another. Differences from OOPM include: (1) there is no concept analogous to DMR, (2) nor is there one analogous to DMML, the OOPM model specification language, and (3) the GUI is not a modeling framework and does not represent multimodel semantics.

3. An Object-Oriented Approach to Integrating Model Geometry and Dynamics

3.1 An Object-Oriented Approach

Classes, objects and relations which form the conceptual model in the OOPM digital world correspond to elements and relations in the source system. This is standard object-oriented methodology. This approach (1) facilitates "object identification," which is capturing elements of meaning that must be represented in the model [19]; (2) is intuitive to model authors; and (3) serves as documentation, which makes a model more self-explanatory to anyone with application domain expertise. Most model authors find at least one of the OOPM dynamic behavior multimodel types to be intuitive and to be a natural way to express behavior of the source system.

During class and object identification, the model author is guided to explicitly recognize the nature of relations among classes. Among these relations are **specialization, generalization** [20, 21], and **aggregation** [22, 4], as depicted in Figure 3. **Specialization** is the relationship of the derived class (subclass) to the base class (superclass). An example from biological taxonomy is that *conch and snail are kinds of gastropod mollusk*. **Generalization** is just the reverse: *gastropod mollusks include whelk and periwinkle*. Specialization often happens when one needs to extend a class in one or several directions; generalization often happens

after the fact, as common natures are recognized and "factored out." Specialization and generalization are associated with inheritance, in which a derived class possesses characteristics of its base class; e.g., mollusks have a foot; therefore the snail, a subclass of mollusk, also has a foot. Coplien [23] recognizes inheritance as a solution domain concept that can be used for subtyping and for code reuse. Subtyping has corresponding meaning in the source system; code reuse does not. Coplien suggests public derivation for subtyping and private derivation for code reuse. Delegation (which implies aggregation) is sometimes better than inheritance for code reuse. In what Coplien calls "forwarding"—a weak form of delegation—a selected subset of constituent class methods are made accessible via methods of the aggregate class.

Aggregation comprises not one but numerous overlapping relations, including **containment**, **composition**, **usage** and **association**, among others [4, 22, 24]. Some examples are a marsh ecosystem *contains* a matrix of patches, a patch *consists of* water and biomass, a snail *uses* sawgrass for food, and a patch is *associated with* climate and hydrology. Sometimes deciding which particular relation applies is problematic; relations should be examined in the context of the source system. As is apparent from the examples above, sometimes distinctions cannot be drawn with certainty, but models can still be elucidated adequately, as long as decisions are reasonable and consistent. Benefit arises from thinking about, discussing and categorizing relations. A reasonable amount of effort spent here is worthwhile; the benefit is as much from the process as from the results. An example of drawing such distinctions is "containment by reference" versus "association by referential attribute" [22]. Both are pointers, and so there is no implementation issue, but the difference is with

regard to lifetimes. In the first case, the object contained by reference should live and die with the containing object; in the second case, the objects have independent lifetimes. This distinction may be important for the model author.

As the model author performs object identification through OOPM Model Author Interface (MAI), a conceptual model is constructed. This mostly visual representation is not unlike the "whiteboard model" mentioned in Section 1, useful for communication with co-workers. Making the classes, objects and relations explicit may help the model author gain understanding. The process may bring to the surface questions and ambiguities that must be addressed to achieve modeling or simulation objectives. When these matters are resolved, the completed model definition is unambiguously and automatically converted to a simulation program in C++. The model author is thus tightly coupled into the modeling and simulation development loop.

3.2 Attributes, Abstract Data Types and Containers

Attributes are defined for each class. In addition to the primitive data types of integer, real and string, OOPM permits user-defined types, commonly known as abstract data types (ADT), to be attributes. ADTs are defined through classes in the model. The OOPM representation of aggregation makes constituent elements attributes of the aggregating class. The best representation depends on (1) cardinality, which is the number of items of a constituent type, such as the 96 patches in the marsh of Figure 4, and whether this number is known in advance and fixed, or is inherently variable; (2) the nature of the relation, discussed in Section 3.1.

Cardinality alternatives include: (1) **"many,"** which causes a container to be created to hold
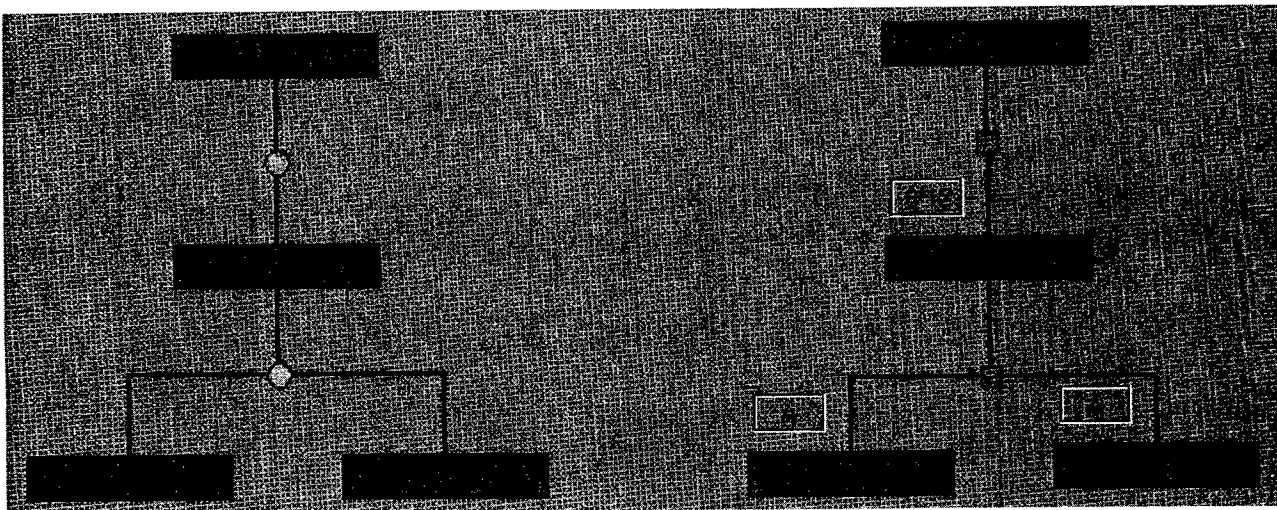

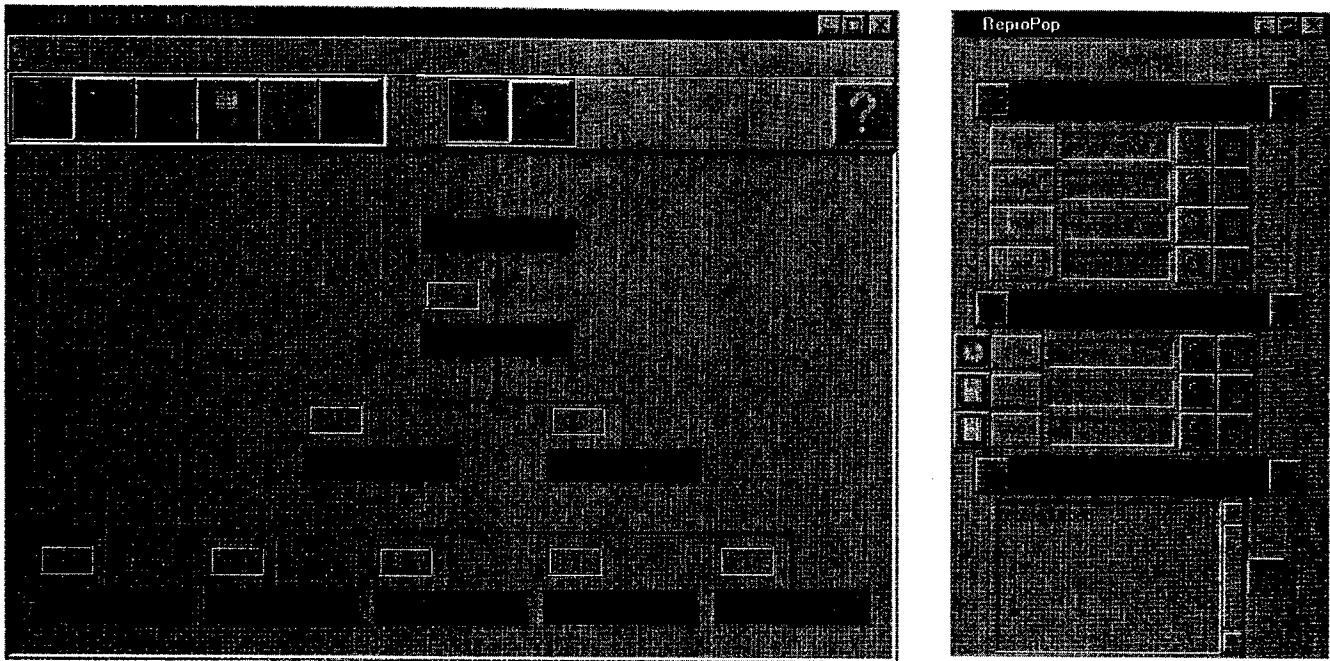**Figure 3**. Relations: specialization/generalization and aggregation

Figure 4. OOPM conceptual model at left; detail of one class at right

contained objects of the constituent class; (2) a numeric cardinality (such as "96"), which also causes a container to be created, but additionally automatically populates the container with the designated number of contained anonymous objects; (3) an "A" is for an association, meaning a referential attribute, which is a reference (pointer) to a named object whose lifetime is independent of the lifetime of the object of the aggregating class; and (4) "V" is for containment by value, which generates a value attribute within the aggregating class.

When cardinality of a constituent class is 1, an ADT attribute will be created in the aggregating class, but a choice remains between value and referential. The "lifetime test" is one decision criterion: if the constituent object lifetime is independent of the lifetime of the aggregating object, then an association, represented by a reference or pointer, is best. It is also possible for the model author to choose a referential attribute when lifetimes coincide; but because value attributes require less management than do referential attributes, value attributes are chosen whenever possible. A second criterion is the "name test." If the object of the aggregated class needs to be a named object created in another part of the model by the model author, a referential attribute, represented by a reference or pointer, is in order, irrespective of lifetime. Yet we have found named objects to be the exception rather than the rule; because names are often irrelevant, named objects force more work onto the model author, and unnamed objects are just as accessible as named objects. OOPM's ability to create *anonymous objects* is quite useful; for example,

1,000 individual models of mobile entities, such as snail kites (birds) in a marsh. The model author considers snail kite objects a fungible collection, and has no need to provide each snail kite with a name, as long as it is somehow addressable. OOPM supports this addressability through containers.

When cardinality is greater than 1, and especially when the number is uncertain, the attribute is a *container class* object, holding objects of the contained type. For example, *SnailKiteS*, a container class, may be instantiated as a value attribute of the marsh class, and may hold 1,000 snail kite objects. Alternatively, a *SnailKiteS* container may hold an arbitrary number of snail kites. Container classes have been found effective to represent an important aspect of aggregation. Provision is made for optional automatic population of containers with a specified number of contained objects or, alternatively, to allow the model author to perform manual population of containers and initialization of their objects. Container classes can be specified directly by the model author, but are usually generated automatically by the cardinality of the aggregation. Containers have behavior which may be extended by the model author: they can send information to their contained objects, execute methods of their contained objects, and select a subset of their contained objects based on some criteria. These features are similar to work of Zeigler [24].

Another aspect of aggregation is how to relate an attribute of an aggregate class to the corresponding attribute in its constituent classes, when such correspondence exists. In contrast to delegation, the problem here is to invoke a method of *every* constituent

class and transform the results into an overall result for the aggregate class. A container known to an object of the aggregate class obtains such information from all its contained objects. The approach is similar to Zeigler's ensemble methods [24]. In the container, all contained objects can be dealt with in the same way using polymorphism. An example is the biomass of snails of several age classes (e.g., eggs, juveniles, adults) in an ecosystem simulation. A snail's biomass is its weight. Total biomass is the sum of the weights of every age class of snail. Moving to a higher level of aggregation, a marsh in the Everglades has a biomass which is the sum of the biomasses of all populations in each of a number of patches, which are areas of the marsh. Here the relation is *summation*, and the common base class has this functionality. A model author is free to specify whatever functionality is appropriate.

4. Facilitating Model Refinement

Modeling is usually iterative and incremental in nature. As the process unfolds, a class hierarchy develops, taking on a tree-like appearance. Levels in this tree are usually related to the level of abstraction which one associates with thinking about and describing the model. Similarly, as dynamic behavior models are specified, using any one of several model types, it may be desirable to refine any one element of a model into another model in its own right. Each element of a model is termed a "block." Examples of blocks in a finite state machine are "state" and "transition." The model within the block may be either the same type or a different type as the type of the larger model containing the block. When one can mix and match model types arbitrarily in a hierarchy of any depth, this is a heterogeneous model hierarchy.

4.1 Coupling

To support an arbitrary heterogeneous model hierarchy, our models must be *closed under coupling*. This suggests that the method of coupling one model component to another must be clearly defined. Two kinds of coupling exist: intralevel and interlevel. Intralevel coupling reflects model components coupled to one another in the same model. For example, one needs to specify rules of how Petri nets, compartmental models and system dynamics graphs are formed. With a system dynamics graph, a rule of model building defines that any level has an input rate and an output rate.

A more interesting case arises in interlevel coupling, because we must ensure that we define rules as to how model components from one model can be refined into models of different types. Can a finite state machine be refined into a Petri net, or can a

functional block model contain finite state machines (FSM) inside blocks? What are the rules to guide this refinement? The rule for interlevel coupling is based on "block decomposition." Each model is defined as a graph, and each graph component is defined as a *block*. This generalizes the semantics normally associated with most components to the extent that each component now maintains the power and flexibility of an object, with its own attributes and methods. For example, an FSM can be our candidate dynamic model. Since it is, by default, expressed as a graph, we take each state and transition and define each as a block. State names become block names and transitions become boolean methods within the block that they define. Also, the FSM itself is a block, so that a model becomes a block defined in terms of connected blocks, which is an architecture that lends itself to recursively-defined coupling.

4.2 Multimodeling

Multimodeling is a recent development [8, 9] which provides multiple levels of abstraction [10] to represent geometry and dynamic behavior of a model. In OOPM, multimodeling permits a variety of popular dynamic model types, including finite state machine (FSM), functional block model (FBM), differential or algebraic equations (EQN), rule-based models (RBM) and system dynamics models (SDM). When these dynamic multimodel types are not appropriate, model authors may create "code methods" for dynamic behavior, or as wrappers, to encapsulate legacy code. Support for a variety of model types is an important intentional departure from the norm. Variety contributes breadth which can accommodate diversity of background and preference in model authors. Breadth is also needed to accommodate variety in source systems and application domains. OOPM has the capability to seamlessly "mix and match" heterogeneous dynamic behavior model types at model definition time, and also to "hot swap" components at simulation runtime. Multimodeling facilitates: (1) model development, selective refinement to achieve required fidelity or model extensibility, and accommodating unanticipated change; (2) integration and reuse of object-oriented distributed simulation models; and (3) extensibility of the framework itself to accommodate future model types.

Because model development resources are limited, one typically refines using a breadth-first approach, and this tree-like structure accordingly takes on an uneven shape, with some parts of the tree being of greater height, and others being shorter, reflecting the underlying decision criterion to refine only as needed to achieve required model fidelity. Development is often iterative and incremental. One usually

takes a model to a simulation, runs it, and uses results to determine where more modeling work is needed. Multimodeling can conserve development resources by providing an orderly framework within which refinement as needed may proceed. A shallow model can be run, and analysis can pinpoint model subtrees where additional fidelity is needed. This adaptive mechanism can focus and guide development. The evolving model is thus its own prototype. It needn't be discarded, as in throw-away prototyping, nor does it suffer the chaos that often accompanies the "exploratory prototyping" or "exploratory programming" approach [3].

The multimodel definition is recursive: refinement proceeds as far as needed. The level of refinement may be bound at model definition time or at simulation runtime. When bound at model definition time, the simulation program will not change its components on the fly. When refinement is bound at simulation runtime, this permits "hot-swapping" of components. For example, refinement of such a multimodel will change on the fly in response to system constraints. A typical constraint is a real-time constraint on when the simulation must complete. Presently, OOPM does not provide the executive logic which decides when to change refinement depth; but, given such logic, OOPM has implemented a capability to reconfigure model refinement on the fly. Others are working on providing the executive logic for this kind of multimodeling in OOPM [25].

5. Visual Elements of OOPM

Visual elements of OOPM include a conceptual modeler, use of VRML for geometry models, several OOPM editors (one for each of five types of dynamic multimodels), and Scenario, which provides simulation runtime output visualization. Supported platforms for visual elements of OOPM include the Solaris dialect of UNIX, Microsoft Windows NT 4.0, and Windows 95.

5.1 Conceptual Models

The OOPM Model Author Interface (MAI) is a graphical user interface (GUI). Modeler relies on the Distributed Model Repository (DMR, discussed later) to store model definitions as they are constructed, and as a source of components for reuse. The main part of MAI is Conceptual Modeler, discussed here. Additional parts of MAI are a set of dynamic model editors, discussed in Section 5.3. A model can be created from scratch, or can be an integration reusing proven smaller models as subsystems obtained from DMR.

The Conceptual Model defines classes, objects, relations among classes, and relations among objects

(aggregation and specialization or generalization). Small rectangles representing classes are arranged, using their relations to form aggregation and specialization/generalization hierarchies. When a small class rectangle is double-clicked, it opens to reveal class detail, including the name of the class, its attributes, its methods and its named objects. Within each method, the model author may specify input parameters, output parameters, return type, and which dynamic model type the method is to be, or whether it is to be a code or constructor method.

5.2 Geometry

Our focus is to apply multimodeling to geometry, as we do with behavior (see Section 5.3). We do not seek to invent a solution for geometry. We prefer to reuse existing solutions that already exist. We seek to provide the framework to allow powerful capabilities of geometry representations such as Virtual Reality Modeling Language (VRML) to be available to the OOPM model author.

Geometry relates objects over a space. The OOPM geometry class library has classes such as Matrix, which represents a two-dimensional grid, like the patches in a marsh, and provides two services: de-referencing and iteration. De-referencing takes two coordinate values and returns the object at that coordinate. Iteration evokes a particular behavior of every object in a set. Matrix is a base class of marsh. This confers on a marsh an ability to manage its geometry. Many source systems and their models fit this geometry metamodel. Other geometry types under consideration for the geometry class library are hierarchy trees, such as constructive solid geometry (CSG) and quad-tree.

In the metamodel mentioned above, source systems typically have free-roaming entities which interact and evolve over a field, with the field influenced and changed by the presence and activities of the entities, defining properties of the space over which the field is defined and through which entities move. Additionally, each spatial unit may contain objects which are fixed to reside in that unit; often a diffusion process is active over the field. This metamodel is descriptive of a wide variety of source systems, from polymer chemistry to ecosystems. An example of mobile entities would be snail kites (birds) in a marsh. An example of a fixed object is the snail population in each patch of the marsh. Diffusion operates along gradients in snail population density between adjacent patches.

Early work with OOPM Scenario was done in the GUI toolkit (Tk) of Ousterhout's TclTk program. This is being supplanted by the Virtual Reality Modeling Language for several reasons, including: VRML is more immersive; VRML is better suited to Web-based
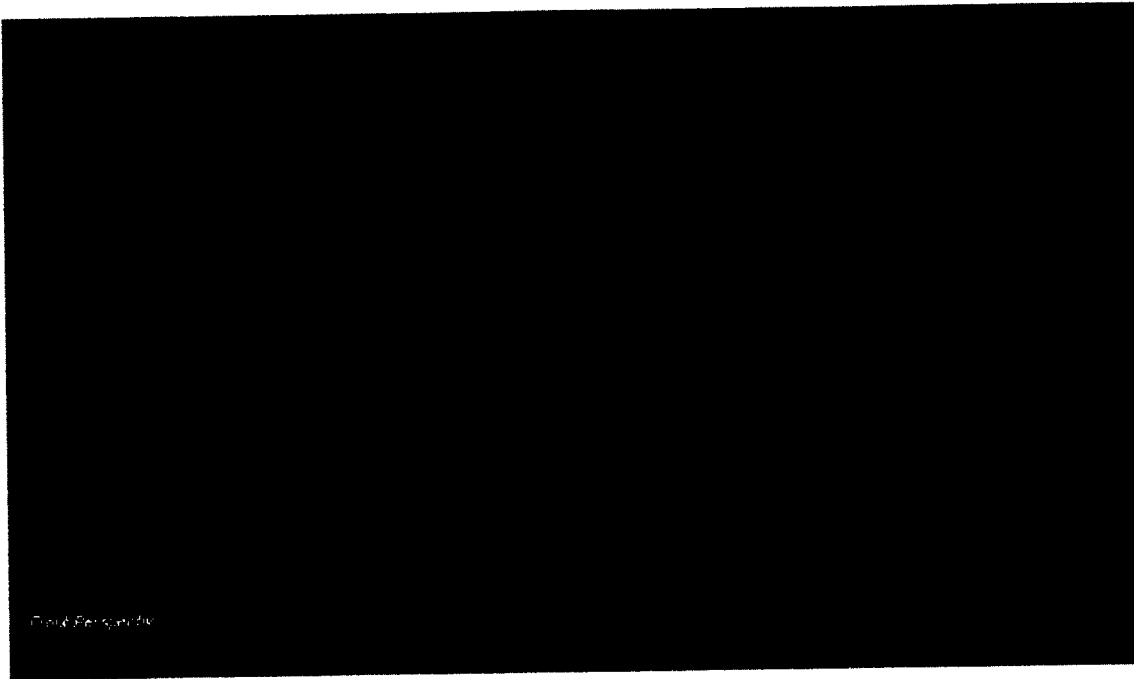
Figure 5. VRML marsh geometry; snail abundance indicated by size of circles

operation than are Tcl applets (Tclets); and VRML has better authoring tools. Each class in a model can have a VRML attribute. Figure 5 shows a VRML model of a marsh, with the size of the circle in each patch indicating abundance of snails in that patch. The affinity of snails for deeper water can be seen in the way the pattern of circles follows the deeper channels in the marsh. In a class which is an aggregation, such as a patch, the VRML model may include VRML models of some constituent classes.

To illustrate this, consider a patch in a marsh, with spatially fixed snail population in the patch, as in the first example. Further, suppose the simulation runtime visualization is relative abundance of three age classes of snails—eggs, juveniles and reproductive adults—in each patch, and over all the patches in the marsh. The three age classes are snail subclasses; each has a VRML model which is a texture-mapped shape representing the age class. The patch VRML model has the three snail subclass VRML models in close juxtaposition, with the size of each shape indicating relative abundance of that group. The marsh VRML model replicates the patch VRML model over the Matrix. Figure 6 zooms in on a few patches of this marsh VRML model.

5.3 Dynamic Behavior

In OOPM classes, dynamic behavior is represented by *dynamic multimodels*, allowing the model author to specify a model (and thus its corresponding simulation program) without being a programmer. OOPM presently incorporates five kinds of dynamic multimodels, each presented below. The model

author is free to use or avoid any particular multimodel type, either because of the diversity of backgrounds and preferences of model authors, or because certain source systems and application domains lend themselves more naturally to one multimodel type than to another. The model author can mix and match various dynamic model types arbitrarily to define methods of the classes of the model. Each dynamic multimodel (1) is created by an OOPM visual editor, (2) involves drawing pictures like the "whiteboard pictures" mentioned in Section 1, and (3) has the rigor of the formalism which underlies the multimodel type.

Every OOPM dynamic model is (potentially) a multimodel, with a structure (subordinate elements), a topology (how those subordinate elements are connected), inputs, and outputs. In OOPM, every subordinate element of every dynamic model (e.g., a state of a finite state machine) is an object of a derived class of a universal behavior base class, and so can in turn be another multimodel of any type, in principle, *ad infinitum*. This not only facilitates model refinement, it also supports heterogeneous multimodels and runtime multimodels (to be discussed later).

Each method M_j of class C_i is a dynamic multimodel of some type. Within $C_i :: M_j$ are subordinate elements. Each such element may be of any method: (1) of C_i; (2) of any value attribute of C_i which is an abstract data type (ADT); (3) of any referential attribute of C_i which is an ADT; or (4) of any associated object. The first two groups are bound at class declaration time. The third and fourth groups permit dynamic binding, and so support

49
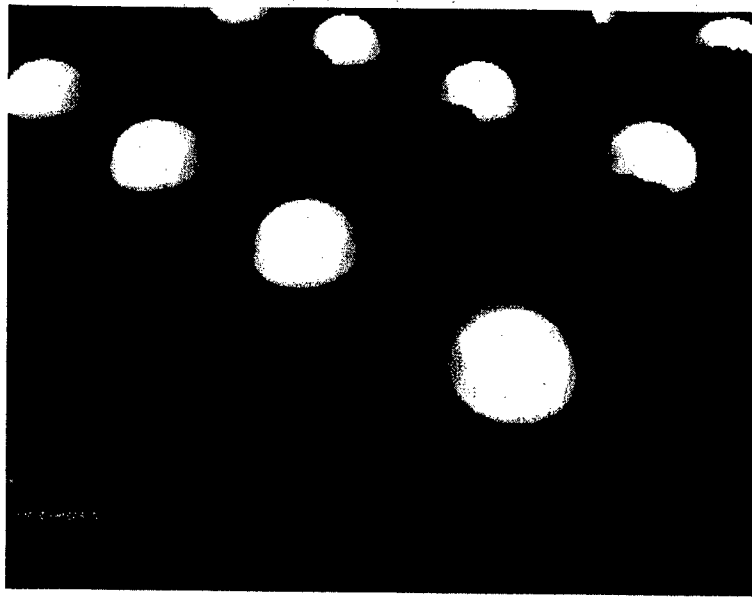
Figure 6. VRML marsh; abundance of three snail age groups indicated by texture-mapped snail models

polymorphism—in which an association to an object of some base class, such as mollusk, can be satisfied by any object of any derived class of mollusk, such as snail—and a (virtual) method call will result in a call to a method of the appropriate derived class corresponding to the type of the associated object (snail), without the specific type of the associated object being known to the calling code. Use of polymorphism permits "hot-swapping" of one model for an equivalent model on the fly at runtime, which supports runtime multimodeling [25].

5.3.1 Finite State Machine (FSM)

The model author designates a method of a class as a Finite State Machine (FSM), and then uses the OOPM FSM editor to construct the FSM. An FSM is a directed graph consisting of states (the nodes) and transitions (the arcs). On each transition appears a
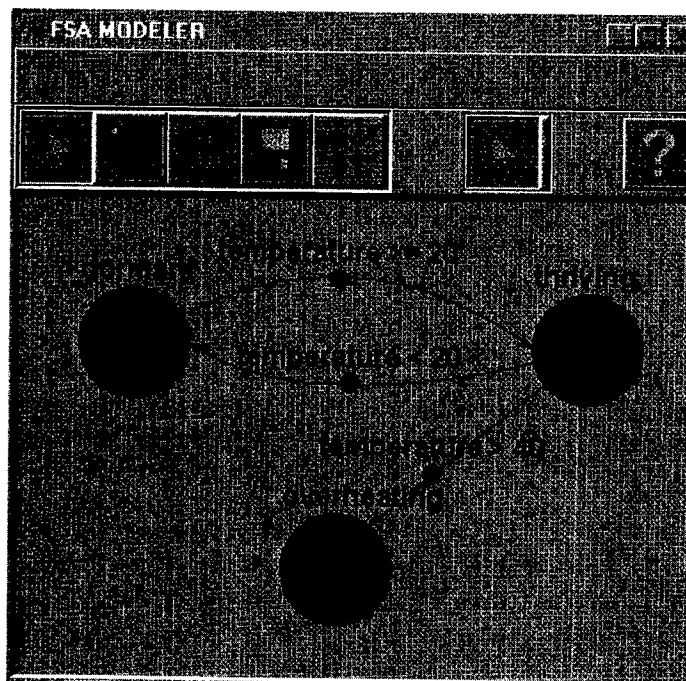
Figure 7. Dynamic multimodel: finite state machine for snail population response to changing ambient temperature

50

predicate. Each state and each transition of the FSM may be another multimodel. At any time the FSM has a current state. Its dynamic behavior causes the FSM to change state from state i to state j when the predicate of $transition_{ij}$ is true. If several transitions have true predicates, ties are broken arbitrarily. An OOPM FSM is shown in Figure 7, representing snail population response to changing ambient temperature.

5.3.2 Functional Block Model (FBM)

The model author designates a method M_j of class C_i as a Functional Block Model (FBM), and uses the OOPM FBM editor to construct the FBM. An FBM has blocks and traces. Blocks appear on the canvas as rectangles, like chips on a circuit board. Inputs and outputs of each block look like pins on a chip. The model author connects various output pins on one block to various input pins on another block. These "traces" form the FBM's topology. Inputs to the FBM, if any, are connected to block inputs. Outputs of the FBM, if any, are from output pins of various blocks. Cycles are permitted, and these propagate a value at one time-step to the next. Several class libraries of pre-written blocks are available, but not required, including "control applications" (Add, Subtract, Multiply, Divide, Integrate, Constant, PseudoRandom and Accumulate), the "queuing model" (Source, Sink, Fork, Join and Facility), and

the "flowchart model" (Begin, End, Decision, Process and Auxiliary).

An OOPM FBM is shown in Figure 8 representing the life cycle of snails. In the forward direction, eggs grow to juveniles, then mature to adults. A cycle is explicitly indicated by the trace from reproductive adult to egg. The block at the bottom with many inputs records results.

5.3.3 Equation Constraint Model (EQN)

The model author designates a method of a class as an Equations Constraint Model (EQN) [26], and then uses the OOPM EQN editor to construct the EQN. An EQN model consists of a system of any number of nth order differential equations, as well as algebraic equations. The syntax is that of C++, and math functions such as $\sin(x)$ may be used. Differential equations are represented using symbols such as x, x' for the first derivative, and x'' for the second derivative of x. Several state variables may appear. The output of the system may be any order derivative of any variable. If a state variable used in the system of equations has the same name as an attribute of the class to which the EQN model belongs, then the attribute and the state variable denote the same entity. Either updates the other as appropriate.

In addition to variables and their derivatives, a set of equations may contain (additive and multiplicative) parameters and input signals. Parameters may
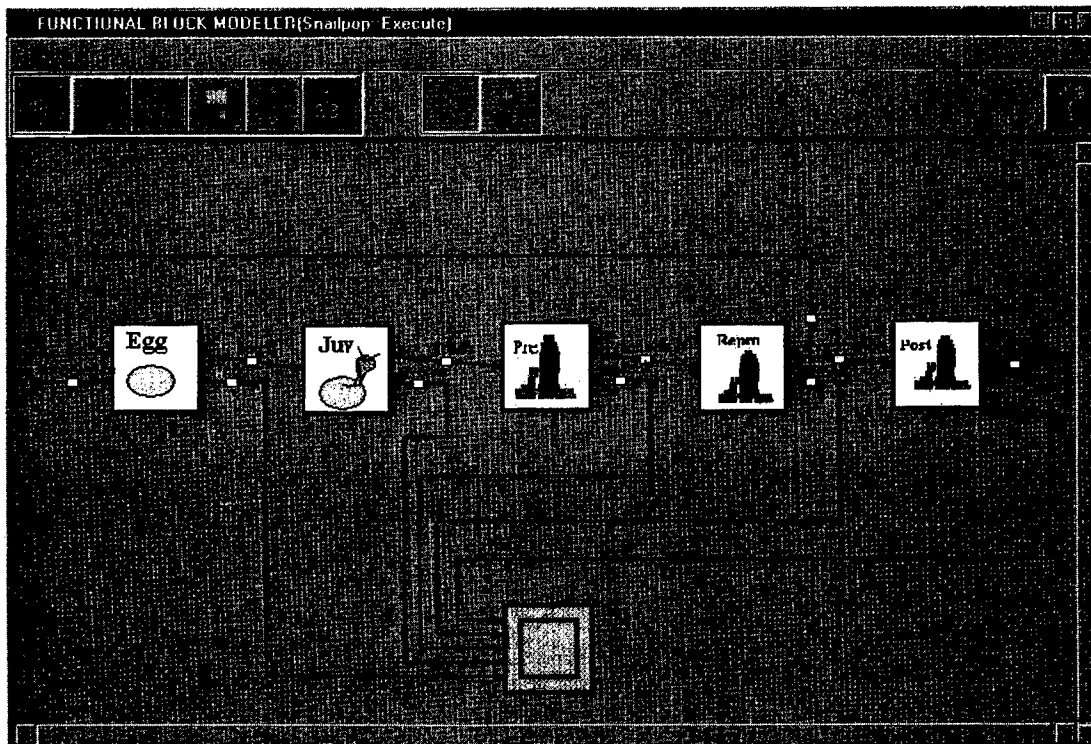
Figure 8. Dynamic multimodel: functional block model depicting snail life cycle

51
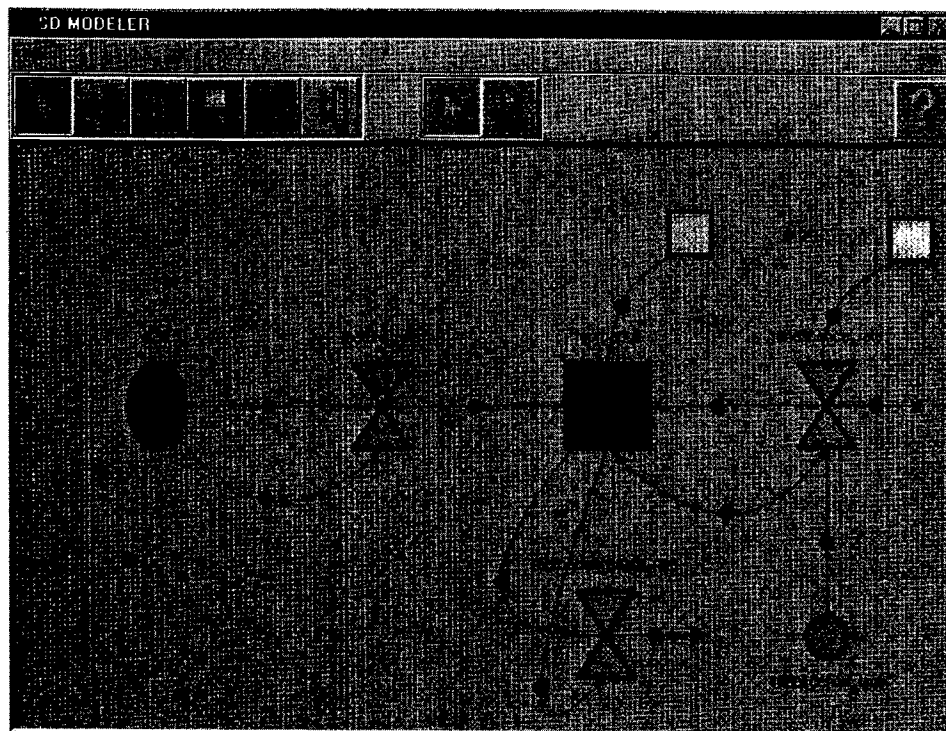
Figure 9. OOPM system dynamics model for snail behavior

be attributes of the class to which the model belongs, or they may be input parameters to the EQN method, or they may be multimodels.

5.3.4 System Dynamics Model (SDM)

The model author designates a method of a class as a System Dynamics Model (SDM) [26, 27], and then uses the OOPM SDM editor to construct the SDM. System dynamic modeling is a functional modeling technique with a variable-based, rather than a function-based, approach. Elements of an SDM include levels, rates, sources, sinks, constants and auxiliaries, as well as two kinds of arcs: flow arcs and cause-and-effect arcs. As with other models, elements may be multimodels.

An OOPM SDM is shown in Figure 9 representing dynamic behavior of a reproductive adult snail population. Rates which affect the population level include maturation, senescence (aging) and death.

The SDM model is equivalent to an EQN model, but some model authors prefer the SDM form. Output of the SDM model editor is identical to output of the EQN editor of an equivalent model. OOPM Translator does not know the difference between EQN and SDM. SDM is the first multimodel type developed in terms of another multimodel type. This approach is an example of reuse which may serve again in the future to further broaden the model author interface while minimizing development effort.

5.3.5 Rule-Based Model (RBM)

The model author designates a method of a class as a Rule-Based Model (RBM), and then uses the OOPM RBM editor to construct the RBM. AN RBM has a set of rules, each expressed as a a conditional expression: if *premise*, then *consequence*. Each premise and each consequence can be another multimodel. The RBM editor has a premise pane and a consequence pane, each of which offers eligible items from lists and for specifying relational and logical operators.

An OOPM RBM is shown in Figure 10 representing snail egg population response to changing ambient temperature. This RBM is a lower-level multimodel within the higher-level system dynamics multimodel described in Section 5.3.4 above.

5.3.6 Code Methods

Although models can be constructed without writing any programs, there may be times when no dynamic model type does what the model author wishes to do. Or, the model author may have a piece of code for a specific algorithm, or some legacy code. For any of these reasons, OOPM permits a model author to write the body of a dynamic model in C++ code, and to integrate that with the rest of the model. The model author interface provides a simple text editing capability for code methods, but the model author is free to use his or her favorite editor instead (any text editor that works with ASCII files).

52

5.4 Scenario

Analysis of simulation execution has often in the past focused on massive amounts of tabular data. Output visualization is effective in facilitating analysis and understanding. Scenario does this. Additionally, Scenario can initialize parameters and pass them to Engine. This is a Model/View/Controller architecture, where Scenario is View and Controller, and Engine is Model.

The new OOPM geometry representation is VRML. Each class may have a geometry attribute, which can be or include a VRML world. VRML authoring tools are external to OOPM; nonetheless, the ability of classes to bring with them their VRML representation is valuable for reuse and integration.

OOPM Scenario is a visualization enabler. Scenario activates and initializes simulation model execution by running the program we call Engine, at the request of the user. Scenario maintains synchronous bidirectional interaction with Engine. In the visualization, Scenario displays Engine output in a form meaningful to the user. In the controlling role, Scenario allows the user to interact with Engine, modifying simulation parameters and changing the rate of

simulation progress. Engine can be allowed to free-run, or can be made to single-step through one event at a time (the default), or to run at any pace in between. As a separate feature, simulation clock time scales can be stretched or compressed. Both can be combined to generate animations with which the model author can interact. Things which happen too fast can be slowed down. The rate of progress can be adjusted to focus on parts of the simulation execution that are of particular interest.

The first Scenario was based on Tcl/Tk. The new Scenario is based on VRML. Both use Transmission Control Protocol over Internet Protocol (TCP/IP) communication between Engine and Scenario. In the TclTk Scenario, communication is between Engine and the TclTk interpreter. In the new VRML-based Scenario, communication is between Engine and a Java applet which resides on a Web page with a VRML browser plug-in (CosmoPlayer 2.0) for the world being executed. The Java applet communicates with the VRML plug-in using the external authoring interface (EAI). Activation of the Java applet and the VRML plug-in are mediated by our DMX Control Panel, which is a coordinator for Web-based operation
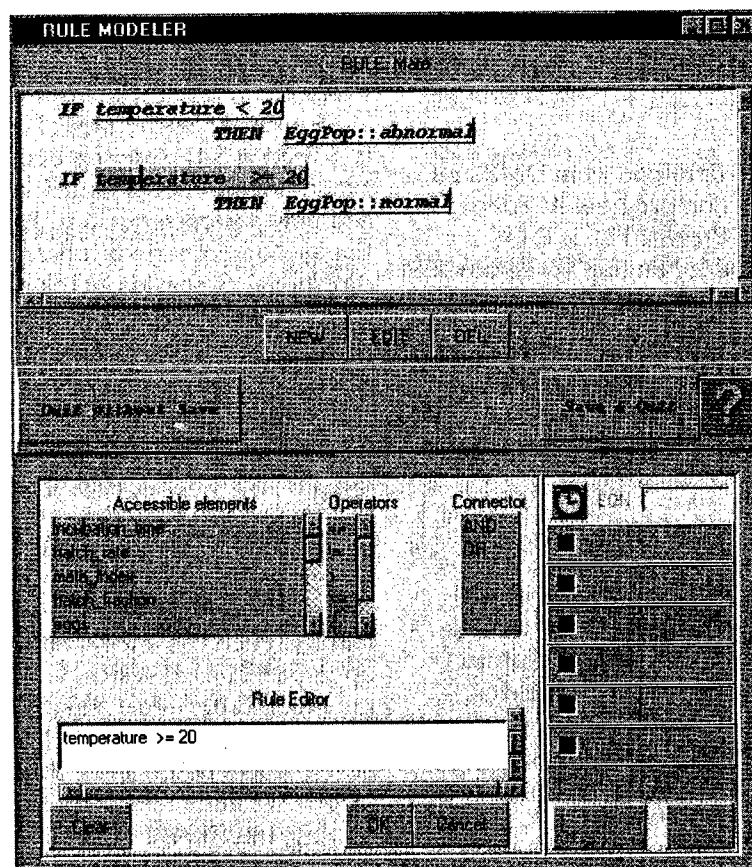
Figure 10. Dynamic multimodel: rule-based model for snail egg population response to changing ambient temperature

of OOPM. This is outside the scope of the present paper, but is described by the authors elsewhere [11].

Scenario detail is unique to each model. OOPM has visualization instruments for reuse, including histograms, xy plot graphs and terrain maps. Nonetheless, some simulation output isn't necessarily amenable to graphical real-time treatment, and there is a necessary role for traditional analysis [28, 29, 30]. OOPM can support this in two ways: Engine can send output for this purpose to output examined by Scenario, or to a separate file. Further analysis can be handled by additional software provided by the model author, e.g., MATLAB.

6. Non-Visual Elements of OOPM

6.1 Introduction

In addition to its visual elements previously discussed, OOPM has non-visual elements, which include Distributed Model Repository (DMR), which holds model definitions; Translator, which maps model definitions to simulation programs written in C++; and Engine, the simulation program including its runtime support libraries. Translator converts a model definition obtained from DMR into a simulation program; Engine is that program. Supported platforms for non-visual elements of OOPM include the supported platforms for the visual elements (Solaris dialect of UNIX, Microsoft Windows NT 4.0, and Windows 95), as well as MS-DOS and OS/2.

6.2 Translator

Translator obtains a model definition from DMR and uses it to construct a simulation program in Translator Target Language (TTL). Present TTL is C++. Translator output is a complete "Engine" program written in C++ including *engine.h*, a header file consisting primarily of class declarations, and *engine.cpp*, a source file containing C++ translation of each dynamic model method and each code method, as well as code to invoke engine runtime support and to synchronize with and accept commands from Scenario.

6.3 Engine

The Engine is the C++ simulation program generated by Translator. It is necessary to compile and link Engine source code to create the Engine executable. This is done automatically using the "make" utility program; alternatively, Engine is compiled and linked using the interactive development environment (IDE) of a compiler such as Visual C++. At link time, runtime support is added from object libraries, the most important of which is *ooSim* [31].

Dynamic behavior multimodels are translated into C++ code, which relies on the underlying event scheduling of the ooSim dispatcher for propagating event

chains. ooSim is event-scheduling simulation queuing model software which is an object-oriented re-implementation and extension of the SimPack toolkit [26, 32]. SimPack is, in turn, based on SMPL [33]. In addition to event scheduling, ooSim provides other support, such as pseudo-random number generation.

Engine source file contains code to initiate one or more event chains. These event chains propagate independently, and the time step of each event chain is independent of the time step of every other event chain. The event scheduler propagates each event chain until that event chain terminates itself, or until the simulation clock reaches the overall time limit specified for the simulation in the model definition. In general, an event chain propagates by rescheduling a specific event routine which the model author identifies. This is accomplished by the *auto_propagate* feature, which is enabled by default. It is also possible for the model author to disable *auto_propagate*, in which case the model itself may generate any number of event chains following any logic. This is an advanced feature which is recommended only to those who are familiar with event scheduling in ooSim and wish to (or need to) have the additional flexibility which manual event scheduling provides. Manual event scheduling is *not* required to get OOPM models to run.

As Engine runs, it executes one simulation event after another, driven by its underlying ooSim Future Event List (FEL). As an event executes, it may generate output. All such output is presented to Scenario (see Section 5.4). After executing each simulation event, Engine checks with Scenario for instructions and new parameter values. The relation between Engine and Scenario is inherently interactive and bi-directional. Scenario can inject events into the FEL of a running Engine. This feature supports distributed execution.

6.4 Distributed Model Repository (DMR)

In the original development of OOPM, model definition persistence was accomplished via textual format, in a set of flat ASCII files comprising a model definition. This approach had (and still has) a number of benefits, including: such model definitions are compact and relatively easy to read, understand and even modify if need be; model definition files get backed up as part of local system backups; models can be put on diskette, into a .zip archive, or transmitted via ftp; it also is a software engineering tool to eliminate development bottlenecks. But this incarnation of OOPM is stand-alone software, with no provision for sharing, thus limiting reuse. Moreover, this OOPM can be used on a machine only after OOPM software is obtained and installed on that machine.

Subsequent progress on OOPM has continued, including: (1) a new approach to model definition persistence we call "model repository," and (2) making the modeling environment Web-based.

The Distributed Model Repository (DMR) holds model definitions, including class declarations, declarations of attributes and methods, and interfaces. DMR provides a database management system (DBMS) for model definitions. Models and model components in DMR are available for browsing, integration and reuse. Class libraries for modeling collections and geometries for spatial models are available. Pieces of a model may reside on different machines, thus permitting model definitions to be distributed, and permitting collaboration within an engineering workgroup on model development.

DMR is more than a DBMS, however, because it transforms information based on multimodeling semantics as data arrives. Model analysis is an integral part of understanding a model definition. An example which occurs whenever a functional block model appears is that each block of the FBM must be examined to ascertain whether it is (1) a method of the class containing the FBM, (2) a method of an ADT attribute of the class containing the FBM, or (3) a method of some other class. Each case is handled differently by Translator: a member method name, a method name qualified with the attribute name, or dynamic binding of a block from the model's context.

In addition to the normal mode of receiving model definition(s) from the model author interface, DMR can also receive model definitions in another way: from text files. These files can be created using a text editor. Historically, such files were originally created by Modeler before DMR existed. These files now serve as a way to initialize, back up or load a DMR.

A reuse example involving DMR is based on the real need to model the snail kite, a bird which is an endangered species and lives in the Everglades. Snail kites eat apple snails, so the snail kite model needs to model the apple snail as a food source. The snail population is heavily dependent on fluctuations in ambient temperature and water depth in the marsh. The snail kite model author is an expert on birds, but not on snails. If she must write her own apple snail model, there will be three drawbacks: (1) its fidelity may be lower than it would be if it were written by an apple snail expert; (2) the complexity of the snail kite model will be greater if writing an apple snail model is part of the job; and (3) development time will be longer. With this in mind, the snail kite model author goes to DMR and learns that an apple snail model has already been written. This apple snail model was written by a snail expert, has been tested and is available. The apple snail model is reused and incorporated into the snail kite model, resulting in a better quality snail kite model, in which complexity is better managed due to the additional abstraction levels, and a shorter development time. Space does not permit us to go into issues such as how the snail kite model author was able to discover the apple snail model, but we have presented this elsewhere [11].

The Distributed Model Repository (DMR) communicates using the connection-based TCP/IP with producers and consumers of model definitions, as an alternative to the local-file-based model definitions mentioned above. Model authors interact with the model author interface, but persistent model definitions reside within DMR; similarly, OOPM Translator converts model definitions to C++ simulation programs, but model definitions are from DMR rather than local files. Benefits include: (1) a model defined on machine A can be translated on machine B; (2) a model can be defined and/or translated on a machine with no (or limited) local persistent store; and (3) models reside where they can best be cataloged, indexed, browsed, backed up and otherwise maintained, without distracting model authors from their primary focus. DMR also permits model sharing in a way that was not available before. For example, a model defined on machine A can be referenced on machines B and C, so A's model is available to B and C, or they can agree to divide the work and pool their results. This not only (4) increases reuse potential, it also (5) provides an environment to support collaborative development. Disadvantages include reliance on network connections and consumption of network bandwidth. DMRs may from time to time start and stop, so the OOPM universe may have any number of DMRs. DMRs know about one another, can forward requests to their peers and can share model information. But DMR does not, in and of itself, make OOPM "Web-based."

Fortunately, there is a way for OOPM to be a Web-based modeling and simulation environment. Our "litmus test" for whether software is "Web-based" is: (1) that it require no installation of separate software, and (2) that it rely on communication conventions of the Web (e.g., URLs). There are several ways to meet these criteria, combining some or all of the following: Hypertext Markup Language (HTML), JavaScript, Dynamic HTML, Java applets, and browser plug-ins with or without LiveConnect. The primary OOPM configuration will be Web-based: a browser plug-in connected via LiveConnect with Java applets, based on HTML with a sprinkle of JavaScript. DMR is a server-side phenomenon, so the standard Web-based configuration does not include a DMR on the client.

7. Plans and Conclusions

We contemplate supporting three additional configurations, one Web-based and the other two not. These

are (1) a Web-based runtime-only (Engine and visualization) configuration; and (2) a "power-user" configuration providing a local DMR and/or a local Java-based GUI and/or a Tcl/Tk-based GUI, which operate out of the same consistent plug-in top level as the Web-based configurations, and which may be more appropriate where network bandwidth and/or security issues are paramount. Finally there is (3) the original stand-alone configuration of OOPM. The last two configurations are not Web-based because they require OOPM and possibly Tcl/Tk to be installed on the local machine.

Distributed Model Repository (DMR) is a substantial step in the right direction. Web-based operation of OOPM will soon be upon us, and the use of DMML as a representation common to all elements of OOPM will tie our architecture together in a way that we hope will have significant benefits for making reuse of object-oriented distributed models practical. We learned that TclTk neither enforces nor facilitates object-oriented methodology, and are working on a Java applet-based MAI to improve reusability and extensibility of our code. Scenario has a difficult job, managing simulation runtime output visualization even though every model is different in surface appearance, but many others have done good work in this area and with VRML, we are confident that the best features of OOPM can be merged with the best features of powerful runtime scenario tools. Dynamic behavior multimodel types now implemented provide breadth, but we are looking at extending further in this direction, for example, Petri Nets. For model geometry, we need to fully apply the same heterogeneous multimodeling which has worked well for us with dynamic behavior. We also continue to look at immersive technologies for the MAI.

8. Acknowledgments

9. References

[1] Johnson, Ralph E. "Frameworks = (Components + Patterns)." *Communications of the ACM*, Vol. 40, No. 10, pp 39-42, October 1997.

[2] Dahmann, J. "Department of Defense DMSO High Level Architecture." *ITEM98: International Training and Education Conference, Palais de Beaulieu, Lausanne, Switzerland*, ITEC Ltd., Warminster, Wilshire, United Kingdom, April 28-30, 1998.

[3] Sommerville, I. *Software Engineering, Fourth Edition*, Addison-Wesley, 1992.

[4] Booch, G. *Object-Oriented Analysis and Design with Applications, Second Edition*, Addison-Wesley, 1994.

[5] Fishwick, P. A. "The Role of Process Abstraction in Simulation." *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 18, pp 18-39, January/February 1988.

[6] Fishwick, P. A. "Abstraction Level Traversal in Hierarchical Modeling." *Modelling and Simulation Methodology: Knowledge Systems Paradigms*, B. P. Zeigler, M. Elzas and T. Oren, editors, pp 393-429, Elsevier North Holland, 1989.

[7] Berzins, V., Gray, M. and Naumann, D. "Abstraction-Based Software Development." *Communications of the ACM*, Vol. 29, No. 5, pp 402-415, 1986.

[8] Fishwick, P. A. and Zeigler, B. P. "A Multimodel Methodology for Qualitative Model Engineering." *ACM Transactions on Modeling and Computer Simulation*, Vol. 2, No. 1, pp 52-81. 1992.

[9] Fishwick, P. A. "A Simulation Environment for Multimodeling." *Discrete Event Dynamic Systems: Theory and Applications*, Vol. 3, pp 151-171, 1993.

[10] Fishwick, P. A. and Lee, K. "Two Methods for Exploiting Abstraction in Systems." *AI, Simulation and Planning in High Autonomous Systems*, pp 257-264, 1996.

[11] Cubert, R. M. and Fishwick, P. A. "Software Architecture for Distributed Simulation Multimodels." *SPIE AeroSense98 Conference Proceedings, Volume 3369*, SPIE, Bellingham, WA, April 1998.

[12] Zeigler, B. P., Song, H. S., Kim, T. G. and Praehofer, H. "DEVS Framework for Modeling, Simulation, Analysis and Design of Hybrid Systems." *Hybrid Systems II, Lecture Notes in Computer Science*, E. P. Antsaklis, et al., editors, pp 529-551, Springer-Verlag, Berlin, 1995.

[13] Zeigler, B. P., Moon, Y., Kim, D. and Ball, G. "The DEVS Environment for High-Performance Modeling and Simulation." *IEEE Computational Science and Engineering*, Vol. 4, No. 3, 1997.

[14] Zeigler, B. P. *Theory of Modeling and Simulation*, John Wiley and Sons, New York, 1976.

[15] Rumbaugh, J., Blaha, M., Premerlani, W., Frederick, E. and Lorenson, W. *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[16] Hill, D. R. C. *Object-Oriented Analysis and Simulation*, p 104, Addison-Wesley, Reading, MA, 1992.

[17] Hill, D. R. C. and Vigor, E. "Simulation and Software Engineering: Bridging the Culture Gap with UML." *SCS Object-Oriented Simulation Conference*, January 11-14, 1998, San Diego, CA, pp 81-87, 1998.

[18] Daconta, M. C., Saganich, A., Monk, E. and Snyder, M. *Java 1.2 and JavaScript for C and C++ Programmers*, John Wiley and Sons, New York, 1998.

[19] Zeigler, B. P. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Proess, 1990.

[20] Stroustrup, B. *The C++ Programming Language, Second Edition*, Addison-Wesley, 1991.

[21] Booch, G. and Rumbaugh, J. *Unified Method for Object-Oriented Development*, Rational Software, 1995.

[22] Riel, A. J. *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[23] Coplien, J. O. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

[24] Zeigler, B. P. *Objects and Systems*, Springer-Verlag, 1997.

[25] Lee, K. and Fishwick, P. A. "Semi-Automated Method for Dynamic Model Abstraction." *SPIE Conference Proceedings*, 1997.

[26] Fishwick, P. A. *Simulation Model Design and Execution: Building Digital Worlds*, Prentice-Hall, 1995.

[27] Rosenberg, R. C. and Karnopp, D. C. *Introduction to Physical System Dynamics*, McGraw-Hill, New York, 1983.

[28] Payne, J. A. *Introduction to Simulation: Programming Techniques and Methods of Analysis*, McGraw-Hill, 1982.

[29] Law, A. M. and Kelton, W. D. *Simulation Modeling and Analysis*, McGraw-Hill, 1991.

[30] Fishman, G. S. *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley and Sons, 1973.

[31] Cubert, R. M. "The OOSIM Object Oriented Simulation Library." Technical Report to CISE Simulation Group, University of Florida, 1995.

[32] Fishwick, P. A. "Simpack: Getting Started with Simulation Programming in C and C++." *Winter Simulation Conference WSC'92 Proceedings*, pp 154-162, 1992.

[33] MacDougall, M. H. *Simulating Computer Systems: Techniques and Tools*, MIT PRess, 1992.

Robert M. Cubert is a PhD student in the Computer and Information Sciences Department at University of Florida. His foremost research interest is in making practical the Web-based reuse of object models. He holds a BS degree in Electrical Engineering from MIT and a BS in Zoology and an MS in Computer Science from the University of Oklahoma. He spent three years on the Computer Science faculty at California State University, Sacramento, and has a decade of industry experience writing software for real-time control systems and data communications.
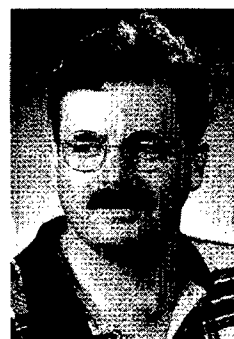
Paul A. Fishwick is an Associate Professor in the Department of Computer and Information Sciences at the University of Florida. He received a BS in Mathematics from Pennsylvania State University, an MS in Applied Science from the College of William and Mary, and a PhD in Computer and Information Science from the University of Pennsylvania. He also has six years of industrial/government production and research experience working at the Newport News Shipbuilding and Dry Dock Company, doing CAD/CAM parts definition research and at NASA Langley Research Center, studying engineering data base models for structural engineering. His research interests are in computer simulation modeling and analysis methods for complex systems. He is a senior member of the IEEE and the Society for Computer Simulation. He is also a member of the IEEE Society for Systems, Man and Cybernetics, ACM and AAAI. Dr. Fishwick founded the *comp.simulation* Internet news group (*Simulation Digest*), which now serves more than 15,000 subscribers. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years and he is on the editorial boards of several journals, including the *TRANSACTIONS of the Society for Computer Simulation, ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics, International Journal of Computer Simulation*, and the *Journal of Systems Engineering*.

Computer Simulation: Growth Through Extension

Paul A. Fishwick

Computer and Information Science and Engineering Department, University of Florida, Gainesville, FL 32611
E-mail: fishwick@cise.ufl.edu

Computer simulation is a fundamental discipline for studying complex systems. Similar to any other discipline, simulation must grow and be fine-tuned so that it maintains its position as the base methodology for carrying out computational science and constructing digital worlds. We discuss ten areas outside of simulation and demonstrate growth by identifying relationships between simulation and each of the areas. We outline each field by describing it briefly and then specifying outstanding issues that remain to be resolved. We have found that we are better able to characterize basic simulation methodology by integrating and extending simulation within the context of other fields.

Keywords: Taxonomy, simulation, model design, simulation survey, state-of-the-art

1. Introduction

The field of computer simulation is approximately forty years old, and it is still vibrant and growing. As technology develops faster hardware, old forms of simulation are made to go faster, and new varieties of simulation emerge through an *extension* process. Extending the core simulation knowledge base involves taking existing simulation concepts and blending them with concepts outside of the simulation discipline. An example of extension is taking two concepts, a system model and an abstract programming *object* (from object-oriented [OO] design), and seeing how they relate to one another. We can extend system models by designing model components as objects. Although this extension seems simple enough, controversies will arise. Should all physical systems be modeled with objects, or are some better modeled with equations, for instance? How do equational models mesh with object-based models? Sometimes, the interface between the simulation concept and the extensional concept is straightforward, but, in most instances, there are many issues to be addressed. The ultimate goal within the simulation community is to walk the narrow line separating a mathematically defined system theory-based foundation, on one hand, and the world outside of simulation that encourages extension and possible revision of the basic approaches.

Extension—when used with simulation—means that we consider an arbitrary topic and then study how this topic can be integrated with simulation. The morphological box concept [1] provides a formal way of studying the interaction between different topics and the field of computer simulation. This box forms a new relation by considering two orthogonal sets and taking the cross-product. The cross-product forces one to study interactions in an organized manner. One possible morphological box is shown in Table 1. This type of focused approach promotes the discovery of new extensions and concepts for the simulation field. It is possible that some cells will be empty, representing that a clear relationship does not exist for that particular row-column combination. This box breaks simulation down into three subfields: *model design, model execution,* and *execution analysis*. Model design reflects how we should design and engineer models from concepts to something that can be executed on a computer. Model execution includes serial and parallel algorithms for simulating the model once it has been designed. Execution analysis uses statistical procedures to collect data and verify and validate models. These three subfields are listed as columns. For each row, we list several fields outside of simulation. As we will see, these external fields serve as vehicles for extension.

Let us consider the entry in Table 1 identified by the area *artificial life* (AL). The relationship between AL and simulation model design is that most models for AL are discrete and spatial in character. That is, the models use types such as cellular automata and L-Systems [2]. To consider the next column—model execution—we will simulate AL models by employing simulated evolution with genotypes and operations such as *mutate* and *crossover.*

Many fields within the purview of computer science and computer engineering serve as candidates for the extension process. That is, we take our simulation knowledge base and create extensions by linking to these fields. The forums for the exchange of information and suggestions for extension are normally found in simulation conferences, but they can also

Table 1. Morphological box for simulation

Outside Areas	Computer Simulation		
	Design	Execution	Analysis
Abstraction	Homomorphism, Multimodeling	Multi-Level Coordination	Hybrid Methods
Artificial Intelligence	Quality, Autonomy		Issue
Object Orientation	Object Design	Object Methods	
Neural Networks	Behavioral Model	Training	Error Analysis
Fuzzy Logic	Behavioral Model	Fuzzy Arithmetic	
Artificial Life	Spatial	Adaptation	Emergence/Order
Parallelism	Message Passing	Distributed Architectures	Deadlock, Load Balancing
Computer Graphics	Animation as Modeling	Rendering	Validation
Virtual Reality	Geometry Based	Physical Equations	Real-time Evaluation
Information Access	Simulation as Information Gathering	Hypermedia	

be found in workshops located in the extension disciplines. An example of the latter was the series of *AI and Simulation Workshops* held during the National Conference on Artificial Intelligence in 1986 to 1990. Another example was the focus on object-oriented simulation during the 1993 Conference on Object-Oriented Programming and Systems (OOPSLA '93). For this article, we have chosen ten fields that have served as bases for extension to computer simulation. Most of these extensions reflect active research agendas found in most simulation conference technical programs. We begin the discussion by touching on the role of simulation using concepts and quotes. This discussion is meant to be general and introductory. Then, we will describe ten extension areas. Within each area, the following items will be addressed:

- Definition of the extension area and relevance to simulation;
- Issues, controversies, and concerns associated with the extension;
- Literature references on simulation researchers working in the extension area;
- Future approaches and forecasts.

2. Conceptions and Misconceptions

There are many questions that simulationists as well as others ask of the simulation field. These questions suggest that simulation is a growing and vibrant field, trying to achieve a cohesive organization of technical knowledge.

2.1 *"What Is Simulation?"*

It is difficult to form a cohesive discipline when there is no widely accepted taxonomy, although significant efforts have been made [3-5]. Let us create a definition for simulation: *Computer simulation is the discipline of designing a model of an actual or theoretical physical system, executing the model on a digital computer, and analyzing the execution output.* From this basic definition, we derive the three previously defined divisions: model design, model execution, and execution analysis, and subdefine them as shown in Figure 1. Figure 1 also includes cross-references to chapters in a recent book [6].

2.2 *"Simulation Is a Tool"*

A tool is something that a researcher uses because it is handy and useful. Simulation researchers should rejoice in this sentiment because, at the very least, simulation models, algorithms, and software are actually being used in the real world. No greater compliment could be offered. To the extent that the word "tool" implies that simulation is not a research area, one should realize that when one calls an area *X* a "tool," this means only that *one is not doing research in area X* but, instead, needs *X* as a resource. One person's research serves as another person's tool.
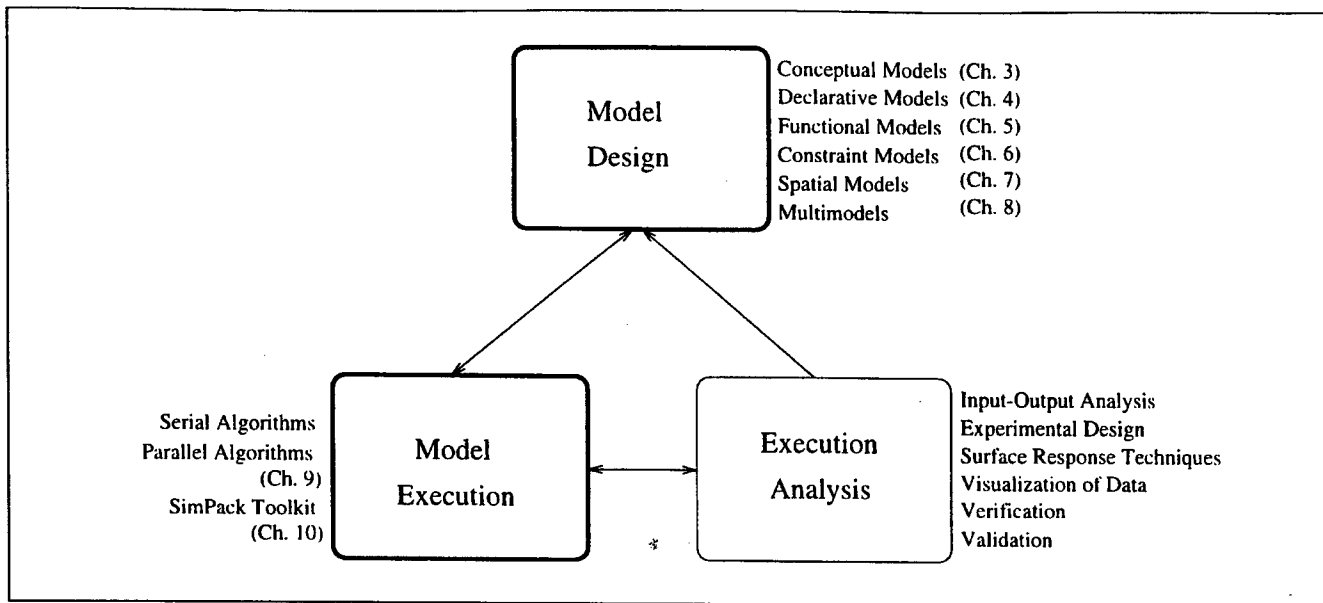
Figure 1. Taxonomy for simulation

2.3 "Simulation of What?"

Simulation, similar to most disciplines, can be generally divided into *methodology* and *applications*. Sometimes, methodology is termed theory. As our field matures and builds on the solid structure of systems theory, we are developing a sound methodology. The importance of methodology—and not just applications—cannot be overemphasized. The simulation discipline has a core of knowledge that is independent of applications. We divide simulation methodology into the subfields of model design, model execution, and execution analysis. Methodology can apply itself to all sorts of practical real-world applications, but it is a substantial field by itself.

2.4 "Simulation Is the Method of Last Resort"

Methods of analytic (non-time-dependent) modeling have been used frequently as a method of first resort because of the expense inherent in the simulation enterprise. Two decades ago, electronic calculators were definitely the computational tool of last resort. After all, they were bulky, expensive, and difficult to find. This is no longer the case, since calculators can now fit on a person's wrist with ease. Moreover, calculators have become easier to use by virtue of their cost. Simulation is in a similar situation. As equipment becomes less expensive and our methods of programming simulations become more efficient (with code reuse and object orientation), simulation will become the method of first resort and, frequently, the only method.

2.5 "Simulations Are Created with a Specific Purpose in Mind"

When one builds a simulation model, it is with the idea of answering a certain set or class of questions about the physical system being modeled. This is the traditional way that we build models and run simulations, but it is in need of an overhaul.

The reason is that, as simulationists, we should be in the market for designing digital worlds containing digital objects. A digital object is one that incorporates all known knowledge about that object so that the object appears and reacts to sensory feedback exactly as the real object would. (Digital object behaviors may be different from real time [faster or slower]). Moreover, a digital object can be asked questions whose answers would have to be at varying levels of abstraction. The idea that we build models to achieve a singular purpose does support analysis by a single user, but we should be building models that are robust and can respond to a wide variety of real-world sensory interactions and queries from many users. A simulationist's responsibility should be to construct the object methods that define how various pieces of geometry, comprising the digital world, react to user intervention. This type of environment will be based on distributed simulation (within the Internet), which will foster code and model reuse and, in turn, will serve as a basis for digital world construction.

3. Model Abstraction: Multimodels

3.1 Discussion

Modeling complex systems requires a "model of models," or a multimodel [6-12]. A multimodel is a network or hierarchy of models in which each model represents the physical systems at a given level of abstraction or granularity. Homomorphic relations formally link each level together, allowing one to traverse levels of abstraction. The need for this type of model was first brought out in combined simulation efforts. Combined simulation focused specifically on blending discrete-event methods with continuous methods, and multimodeling provides an object-oriented methodology to extend this integration so that many additional model types can be integrated. A multimodel is capable of answering a wide variety of

queries and responding to a number of sensory cues (or inputs). In this sense, the multimodel provides the technology for making a digital object, which contains all geometric and dynamic modeling information associated with that object. In addition, the multimodel is computationally more attractive than the single-level model because the analyst may weave through the abstraction network while focusing the computation or dynamics only in those areas that require additional computation. An example of this focusing can be seen in cockpit simulators that use a large projection screen on which the pilot focuses during a simulated flight. The center of the screen—in line with the pilot's foveal vision—uses coarse computer graphics rendering techniques since the peripheral vision is less acute and in need of less graphical detail. For the same reason that we modify rendering complexity, we can also manipulate the complexity of the dynamic model used in peripheral vision, thereby reducing the complexity of the simulation.

3.2 *Issues*

- *Is there a need to simulate levels independently?* In most cases, given a hierarchy of models, it will be sufficient to execute the lowest level model while allowing reporting (model output) at all levels. In the hierarchy, a model can be "cut out" of the multimodel but then we must deal with internal events that serve as input to the model. Normally, these inputs come from the next lower abstraction level.

- *If we have the lower level model, why do we need the higher level models?* Since models represent a human language for exchanging information about dynamical systems, removing the higher level models also removes the more abstract system knowledge. The abstract knowledge serves as an important repository when we wish to reason about system behavior.

3.3 *Future*

To make models more robust, we need to have models containing more than one level of abstraction. Such a model may be more complex, but it can answer a larger class of questions than a single-layer model. We do not always want to see the lowest level of detail for all parts of a process. Moreover, we want to be able to tell the simulation what parts are of interest to us by *tuning in* on those parts of the multimodel. The original term *combined simulation* should be replaced by the more general *multimodel* concept, which fosters an integration of basic model types, as shown in Figure 1.

4. Artificial Intelligence

4.1 *Discussion*

There are two aspects of artificial intelligence (AI) particularly important to simulation research: using natural language and qualitative knowledge, and encoding the decision-making process [13–18]. Humans speak and write in natural language; however, there must be a translation process if this knowledge is to be useful to simulation. Most simulations of natural or artificial systems are based on quantitative methods. In many instances—especially in areas such as social science or medicine—model components (parameters, state variables, input and output) are defined in natural language or in another qualitative representation. There needs to be a way to map quality to quantity. Fuzzy set theory is a well-formed discipline for mapping quality to quantity.

Inasmuch as AI methods can be used within a particular model design, they are even more useful in modeling the decision-making process that envelops the simulation process. Simulations are often used in decision making, and expert systems can be used to guide which simulations are to be executed, and what parameters are to be chosen. The major lesson we can learn from AI is that all knowledge about a system should be encoded—not only that particular knowledge which is quantitative or amenable to analysis. Expert systems provide a good illustration of codifying *meta-knowledge* about a system, and not only low-level aspects of the system.

4.2 *Issues*

- *Psychology or engineering?* First, we must decide whether the goal of the knowledge-based simulation is to validate common-sense human thinking about a system or to validate a physical system whose model was created via a more compiled knowledge approach to human thinking. These are two distinct goals. It is straightforward to create a humanlike model of a four-stroke gasoline engine that is physically inferior to the quantitative model although the model may represent how a particular human thinks about the engine. Only a sturdy experimental method (routinely performed in psychology) can validate a model of human thinking. Conversely, if the model is to represent a physical system, then it must be compared with and contrasted against existing physical system models for a domain.

- *Rules or mathematical models?* When applying AI technology to simulation, at what level is AI most appropriate? Although AI methods can be used to create system models, its primary contribution is at the higher level of organizing the knowledge that makes up the assumptions in modeling and encoding the decision-making process (often using rules) used to control simulation runs. For every system, we must question whether it is appropriate or relevant to use rules, equations, or graph-based models. An arbitrary choice of modeling technique can be problematic. Often the answer to the question of level is to strive to manufacture multimodels, thereby achieving the benefits of each level with its own granularity and definitions for mapping.

4.3 *Future*

We must connect common-sense knowledge to the more compiled, detailed knowledge available for dynamical systems. Having common-sense models of the world that are disconnected

from existing, more detailed models is counterproductive. We need to develop ways of building qualitative models that are demonstrated to be valid based on what we know about a system. Ambiguous data should be represented in the greatest degree of detail possible. Good first steps to defining a variable's value are to use fuzzy sets and probability distributions (if a sufficient sample is readily available). Expert systems should be built to guide simulationists as to what type of model to use in a particular circumstance. Right now, we have very few guiding tools in engineering our models.

5. Object-Oriented Simulation

5.1 Discussion

On one hand, we have the real world, which is full of objects and interactions, and on the other, we have a computer program. A central goal in computer simulation is to map one to the other [6, 19-23]. The most straightforward way of doing this is to create abstract objects in the programming language, where these objects map directly to real-world objects. This approach was first developed in the Simula language and has gained much greater momentum over the past five years. One reason for the lag in OO-based design is that no good visual analog existed for representing class hierarchies, objects, and object interaction. The past five years have produced good visual OO techniques, mostly from the software engineering community. In addition, these OO visual representations can only recently be exploited using recent window-oriented user-interface construction kits.

5.2 Issues

• *Processes or objects?* Should we really be focusing on objects, or should we think in terms of processes and activities? The two concepts are not mutually incompatible: it is natural to create declarative state transition models as an object's behavior; however, the objective way of looking at the world seems most natural. Declarative model components may be defined by refinement into functional models, or vice versa. When looking out of a window, we see tree objects with branch subobjects swaying in the breeze. We usually do not first see the lumped state called "swaying branches." Instead, the concept of swaying is located within an object as one of its methods.

• *Simulation in software engineering.* Many of the examples given in recent OO-based software engineering texts appear to be simulations. Therefore, there is an intense cross-fertilization occurring in this extension area. Programmers are finding that it is easier if we create real-world metaphors for programming tasks, and then construct programs using these metaphors. For example, instead of writing a program to sort *n* numbers in an abstract manner, let each number represent a physical file folder in a filing cabinet. Then create a simulation that allows the programmer to create cabinet, drawer, and folder objects while specifying the sorting procedure as a method available within the drawer

object. As a result, the task of programming becomes less abstract and more attuned to real-world objects. Since simulation is founded on the study of real-world objects undergoing change, a natural confluence now exists between OO-based design and simulation model design.

5.3 Future

We code simulations on a computer using programming languages of some sort. It is natural to want our programming devices to map clearly to the physical-world devices, and so object orientation has many advantages. With the multimodel extension to object orientation, objects can have several abstraction levels. Concepts from distributed simulation provide us with digital objects, which are located with their counterpart physical objects. New worlds are created by choosing the objects that are needed from wherever they are located on the Internet. Objects are then glued together using network messages.

6. Neural Networks

6.1 Discussion

Two approaches have been used for applying Neural Network (NN) research to simulation: (1) the use of an NN as a behavioral model to map a systems input to its output regardless of the nature of the system, or (2) the use of the network as a model of brain activity and human behavior. The first approach involves NNs [15, 24, 25] as repositories of behavior for any system, whereas the second approach presupposes that the system under question is an actual brain whose model is to be validated against empirical data obtained through experiments with a human subject.

6.2 Issues

• *If we know the model, do we need the NN?* Consider an equational model of a system, such as the heat or wave equation. We could also capture the essence of the wave equation, without keeping the equational model, by training an NN to store input-output (i.e., behavior) pairs. Do we want to do that, however, if the model already does this more economically? First, the issue of complexity must be addressed: Which modeling method is faster? (The issue of speed applies both to the time taken to *design* the model and the time taken to *execute* the model.) More importantly, NNs may be useful to *control* systems whose internal state-based model is difficult to discern or obtain. Therefore, if we have an incomplete level of knowledge about a system, the NN approach becomes more appealing.

• *What can humans understand from an NN?* A chief criticism of NNs, which really applies to all behavioral models, is that humans do not gain insight into the way in which NNs perform their internal function. That is, because of a lack of states and events—which are understandable to humans because of potential state/event mappings to natural language—humans are left in the dark. Adequate system

control may be achieved, but to what end? Can (or should) we create systems that we cannot understand? Some recent work attempts to aggregate symbolic knowledge from NNs so that humans can gain insights into NN operation. A reasonable solution is to use NNs as *first-cut* models before a state-space model has been formulated.

- *What about other behavioral models?* The process of using a generic model formulation and fitting values for parameters is known as system identification. We should also ask, then, where we could use nonlinear or linear regression to store input-output system behavior? In addition, to what extent do the NN parameters (such as hidden layer makeup, biases, and starting weights) need to be tuned to make the NN work while minimizing the error?

6.3 *Future*

Neural networks are good *first-cut* models for systems for which we are lacking information, especially in the relationships among state variables. More work should be done to link NN behavioral models to state-space models as they are developed. The current problem is that NN models for systems are not related to other existing models for the same systems. We need to tie them together.

7. Fuzzy Logic and Arithmetic

7.1 *Discussion*

Fuzzy logic is similar to neural networks in that one can create behavioral systems with both methodologies. A good example is the use of fuzzy logic [24, 26–28] for automatic control: A set of rules or a table is constructed that specifies how an effect is to be achieved provided an input and the current system state. The idea of fuzzy logic is to approximate human decision making using natural language terms instead of quantitative terms. Although fuzzy logic creates a behavioral simulation model, fuzzy arithmetic can be blended with classical state-based models. Using fuzzy arithmetic, one uses a model and makes a subset of the system components fuzzy so that fuzzy arithmetic must be used when executing the model.

7.2 *Issues*

- *When should we use fuzzy sets?* Fuzzy logic and sets have been at the center or many debates. Usually, the debate rages about the question of whether probability theory can replace fuzzy set theory. For simulation, we should be concerned about whether there exists any statistical data for a given variable. If data exist in sufficient quantity, there is less of a need for using fuzzy sets, but fuzzy sets may be useful for assigning qualitative values to variables that are less well-defined.

- *Does industrial implementation breed research acceptance?* Fuzzy logic controllers are appearing everywhere from cameras to washing machines. Is this the true test of fuzzy sets— that they have proven themselves in the form of a consumer product, and therefore have a solid foundation? If, by us-

ing NNs, a product can be made more efficient, then, yes, industrial implementation does breed acceptance of fuzzy controllers (or NN controllers for that matter).

7.3 *Future*

Fuzzy sets will be used for the same reason as NNs: as behavioral models of a system which are easy to create, without having to perform a more complicated system identification procedure.

8. Complex Systems and Artificial Life

8.1 *Discussion*

Our first topic [29–32] of complexity is chaos or the study of nonlinear dynamics. We learn from this discipline that models with simple structure can often lead to chaotic behavior when simulated. Simulation is the only real way of studying these systems. Some analytic approaches may be used to obtain rough qualitative features of the chaotic attractor and its component basins and separatrices; however, by simulating these models, we are able to, with precision, numerically determine the basins of attraction and other qualitative features of interest. Analysis breaks down and simulation is the only viable tool remaining.

Our second topic relates to systems composed of very large numbers of homogeneous particles, bodies, or cells. Sometimes the cells can be different, but usually they have a similar structure. Examples of these types of systems are cellular automata, Ising models, Boolean networks, percolation lattices, and *N Body* models. Again, simulation is the only viable method for studying these systems. We can achieve qualitative insight through iterative quantitative means. The field of *artificial life* has sprung up as a branch of theoretical biology. This field represents a bottom-up investigation of complexity using the computer as a type of scientific laboratory. This bottom-up approach to understanding nature is not found only in AL, but also in other complex system model types such as lattice gases for fluid dynamics modeling. For computational fluid dynamics, we can use the top-down approach of starting with the Navier-Stokes equations, or we can approach the problem by starting with elementary conservation laws expressed in cellular automaton rules.

8.2 *Issues*

- *Is artificial life a science?* The AL area could be criticized because of its bottom-up approach to understanding the nature of life, as opposed to the more traditional scientific approach of running experiments on real life forms. Conversely, AL uses simulation as "computational science" by using the computer as a laboratory tool. We cannot understand complex systems without simulating them, just as we cannot understand nature with operating upon it. The models have a life of their own and are no less complex because they were created artificially. One way of viewing the work in AL is to make a comparison with physics. The relationship between AL simulation and the science of biology is similar to the relationship between theoretical

physics and experimental physics. Theoretical and experimental work complement each other.

- *Simulation as the ultimate laboratory tool.* The falling prices of hardware and the increasing costs of performing experiments with real humans and objects cause many experimental methods to become prohibitive. We must ensure that we are not deviating too much from reality and traditional experiments, but we must also embrace a new way of doing science through simulation.

8.3 *Future*

Theoretical studies of complexity, including AL, will continue to grow, since simulation has become more effective because of technological advances in fast computer architectures. Theory should be balanced carefully with experiment. We must be wary of simulations, though, that demonstrate an artificial system for which valid simulation models exist. For example, we could build a spatial model of generic insects with insect behaviors without doing a validation. But what does this demonstrate? If the insect model has not been shown to be physically valid, it should be shown to have demonstrated an important theoretical contribution such as replication, or as a generator of qualitatively distinct spatial patterns, for instance. The purely theoretical AL systems can still be useful, but we need to temper our enthusiasm with attempts at some sort of validation where possible.

9. Parallel and Distributed Computing
9.1 *Discussion*

Simulation usually places a substantial load on the computer on which the model is executed [33–37]. To speed up simulation runs, we can parallelize the model. The vast majority of models to be parallelized are spatial: lattice-oriented automata or partial differential equations. This is because it is relatively straightforward to parallelize over the *domain* (i.e., two- or three-dimensional [3-D] space). Functional models can also be parallelized by using conservative or optimistic approaches. Conservative methods ensure that causality relations among logical processes (functions) are not violated, whereas the optimistic approach assumes that messages arriving first have lower time-stamps. In the event that causality is violated, the logical process states must be *rewound* or *rolled back.*

Aside from the speedup advantage of applying parallel computing technology to simulation, simulation models can also be distributed over a wide area network. One consequence of distributed models is speedup as for the parallel case; however, distributed models are usually associated with real-time interactive simulations such as those used for training purposes and combat simulation. Distributed Interactive Simulation (DIS) is a substantial research project sponsored by the U.S. Department of Defense to allow heterogeneous simulators to communicate on a virtual battlefield.

9.2 *Issues*

- *Network performance in DIS.* There are several problems with running simulations over a wide area network. The key problem is to reduce the network load given a fixed bandwidth. The *entity state packet* is the element that contributes the most to the load problem, so dead reckoning is used to minimize the number of entity state protocol data units that must be issued whenever an entity moves or changes its orientation.

- *Where is everything stored?* In DIS research, it is not clear where to store all of the simulation information, such as the terrain and vehicles. An entity has its own state information, but with dead reckoning, the entity also has the state information (at some level of detail) of a selection of other entities within some radius. Should every entity have a complete map of the terrain? Should there be central simulation servers or should a strict distributed approach be mandated?

- *Extending DIS for all simulation.* The work in DIS suggests that we build a standard for communication among all distributed simulations, and not only those used for combat simulation. The development of new public domain DIS tools will be a thriving research area for the future.

9.3 *Future*

Distributed simulation will certainly speed up our simulation runs; however, it is also likely to change the way we think of simulation models and the composition of such models. We need to start thinking of ourselves as digital world builders instead of workers building isolated models useful to a small number of people. Many parts of the digital world (methods and geometry) will be reused using distributed simulation on the Internet. Object geometry and methods will be physically located where their physical object counterparts are located. For instance, if I want to build a digital world that includes an automobile traffic network, I will use automobile objects located on-line within the automobile manufacturer's object database. It makes little sense to reinvent object geometries and methods for every simulation. If your digital world contains lathes, then that link in your distributed simulation will point to digital lathe objects stored in the database of the company that makes lathes.

10. Computer Graphics
10.1 *Discussion*

Simulationists regard validation to be of critical concern for trusting a mathematical model of a system [38–41]. This concern is well founded since some model structures or animations may "look good," while not being true to the physical phenomenon being modeled. Still, the use of graphics and immersive interfaces is of major importance to simulation since it brings more people into the field. Techniques and tools in computer graphics, such as new rendering methods, endow simulations with the ability to communicate complex behavior in

terms easy for humans to understand (visual communication). The area of *physically based modeling* is of particular interest to simulationists. Simulationists have always used physically based models; however, graphics researchers needing to perform animations have often relied on the multitrack, keyframe approach that is simpler and faster. With the onset of faster machines, graphics researchers are moving to the physically based approach since it yields more realistic animations. The goals of computer graphics and simulation have traditionally been different: the simulationist seeks valid models, and the graphics person looks to create entertaining animations. The lines are not clearly drawn anymore. Since simulationists employ graphical rendering methods, and animators use physically based modeling, there is a convergence of interest. The movement of physically based modeling can be viewed in the larger sense as a movement to do system modeling. Keyframe animations are also models, albeit simpler discrete-state or -event–oriented ones.

10.2 *Issues*

- *Can simulationists use animation techniques?* Computer graphics is moving in the direction of using more system-oriented models, but can simulationists use keyframing methods? (A keyframe is defined as an event in systems terminology.) At first, this may seem contrary to the aim of simulation: validation. Keyframe models are valid; however, they are system models defined at a high level of abstraction. Looked at from this perspective, spline-based keyframing techniques can be used within multimodel simulations in those instances where speed is more important than visual or statistical accuracy. Validation and accuracy need not be systemwide, since the analyst may be focusing only on a small part of the multimodel.

- *Icons or rendered scenes?* Many simulations will use iconic displays instead of fully rendered 3-D scenes. There are two reasons for this: 3-D rendered scenes are computationally expensive, and icons may express the necessary *information content* not contained in the scenery. The ideal situation is one in which the computer doing the simulation is powerful enough to fully render a 3-D geometric model of the system, while also having the capability to display other sorts of information (numeric, iconic) within the 3-D context.

10.3 *Future*

Although many simulation outputs are currently iconic, all future simulations will be based on 3-D geometry and advanced graphics. You will start with the geometry and assign methods and outputs to the geometrical objects.

11. Virtual Reality

11.1 *Discussion*

In the same way that computer graphics is reducing the man-machine communication bottleneck for our model designs and their executions, immersive interface technology [42–44] will impact the way that we physically interact with the computer for model design and execution analysis. Whereas computer graphics focuses on a particular aspect of man-machine communication (i.e., visual feedback), *Virtual Reality* (VR) explores the way in which man and machine can be more harmoniously coupled so that users of the computer feel as if they are immersed in the digital environment rather than being separate from it. We need to investigate how our discipline will change when modeling, analysis, and execution are performed with immersive interfaces. Let us take each of these three simulation topics in turn. Object-oriented models, since they map to physical phenomena, will mesh nicely with the geometrical objects present in the system. Therefore, the modeler will build the model to blend with the geometry. Analysis will not involve simply a table of statistics. Instead, the analyst will "touch" an object and be presented with several ways of obtaining analytical results. By pointing to or touching a server object—with an interest in throughput—the object's color or transparency may change. If numerical results are desired, these statistics can appear as being physically attached to the server. Finally, model execution will involve the analyst being part of a dynamically changing digital world. The analyst can observe the world from a distance or become part of it—becoming one of the objects that is undergoing transition.

11.2 *Issues*

- *The model behind the interface.* VR has received substantial coverage in the popular press and in new research-oriented publications. VR is often viewed as consisting of the hardware man-machine interface issue. This definition is far too limiting, however, and does not reflect the breakdown of research areas required to make VR work. Simulation is needed to drive and respond to the interface. The technology of VR and the science of man-machine communication will require more complex simulation models that respond to a variety of sensory cues.

- *Price/performance and resolution.* Unfortunately, all but the most expensive immersive interfaces lack suitable performance. Although equipment cost is not a technical issue, it affects how much effort simulationists can expend in linking models to humans through more effective interfaces. The resolution of a device is critical if it is to be linked to a simulation. Two major technical hurdles are *lengthy tracking delays* and *coarse resolution* in helmet-mounted displays.

11.3 *Future*

VR represents the future of simulation; however, VR still has too much of a buzzword status. To be successful in VR, we will need improvements in several basic research disciplines, including simulation, man-machine communication, and computer graphics. We must be careful not to let VR become associated solely with man-machine interaction. It represents a much larger movement.

12. Information Access

12.1 *Discussion*

Hypermedia [45, 46] is the marriage of hypertext and multimedia, where multimedia includes documents that contain text, images, audio, and video. Simulation will play a major role in hypermedia research because many of the existing links that reference video and audio files can be defined more generally as programs that produce audio and video as output given user input. Consider opening up a document that describes a new automobile. After reading some textual material on the automobile, the user clicks on a picture of an automobile and is presented with a menu with the following choices: (1) view automobile, (2) drive automobile, or (3) see performance statistics. Item 1 would result in a video of the automobile on the outside and inside depending on where the user directed the viewing using a data glove or other input device. Item 2, however, would result in a simulation that would immerse the human in a realistic driving experience. The key point is that simulation is a natural *part* of information access and not just the generator of information. To weave simulation procedures into multimedia documents will require the ability of hypermedia products to accept input in a formlike manner (or via a VR-type interface) and execute arbitrary programs or scripts to engage the simulation. Distributed simulation will play an important role in hypermedia document retrieval since it is most likely that large complex, but partitionable, models will be distributed, requiring the model execution to be distributed as well. If there are a sufficient number of readers of the documentation, real-time distributed interactive simulation will also be possible while browsing or searching for information.

The World Wide Web (WWW) is a network of hypermedia documents that are located on the Internet. Therefore, WWW sits on top of the Internet, providing a hypermedia information infrastructure. Client programs such as Netscape allow the user to view the documents containing text, audio, and video. With Netscape's introduction of forms, users can use WWW as an interactive testbed and not just a means for obtaining static information. For simulation, the interaction is critical. The tools are in place today for embedding simulation models and their outputs into the WWW, and it is quite possible that WWW-based simulation will become the most predominant mechanism for running any simulation. After all, a user will generally begin the man-machine interaction process by obtaining bits and pieces of information. Many queries will result in static data transfers, but a growing number of queries will necessitate simulation and the use of active data constructs. Database-centered simulation approaches are also important to simulation since the models will have to be stored and retrieved in a logical manner. Object-oriented languages, for the most part, do not incorporate the idea of object persistence, and therefore, do not support object-based structures in an independent fashion. Object-oriented databases will achieve this purpose.

12.2 *Issues*

- *Simulation protocol.* How is simulation information to be transmitted over the WWW? The existing standard for DIS provides some good ideas for packets and their constituents. The information exchange methods used by client programs such as Netscape and hypermedia-based electronic mail programs that incorporate MIMEs (Multipurpose Internet Mail Extensions) can be used as a basis for future hypermedia communication. One approach is to extract concepts in DIS that are generic enough for any type of simulation and then use the MIME format and WWW as a foundation for further development.

12.3 *Future*

Simulation is an integral, fundamental part of information access. For too long, information has been seen as being static in the form of text and images. With the addition of video and audio, we now see that information can include time-dependent information. A natural step in this direction is to have underlying processes producing the video and audio sequences. This production is achieved by using simulation.

13. Conclusions

By combining computer simulation with other disciplines, we obtain extensions that are used to better solidify the current foundation for simulation methodology. We have presented ten fields and their relationship to simulation, along with some current research issues and citations to the literature. It is important that we relate our work to other fields on a continual basis. Without these relations, we can move in the wrong direction or miss a vital convergence that is occurring in other related fields. As it happens, simulation is repeatedly seen as a foundation for many other fields such as those presented herein. There is still quite a bit of work to be done in better organizing the simulation field into the three aforementioned subfields of design, execution, and analysis. The plethora of modeling methods must be contained and categorized so that we can restore some logic to simulation methodology. A constant push-pull process between extension and integration is necessary and will move simulation into the forefront as a core discipline for creating digital world representations.

14. Acknowledgments

15. References

[1] Zwicky, F. 1966. *Discovery, Invention, Research Through the Morphological Approach,* New York: Macmillan.

[2] Prusinkiewicz, P. and Lindenmeyer, A. 1990. *The Algorithmic Beauty of Plants.* Springer-Verlag

[3] Zeigler, B.P. 1976. *Theory of Modelling and Simulation.* New York: John Wiley & Sons.

[4] Ören, T.I. 1987. "Simulation model symbolic processing: Taxonomy," *Systems and Control Encyclopedia,* Singh, M.G., ed., Elmsford, NY: Pergamon Press, pp. 4377–4381.

[5] Ören, T.I. 1987. "Simulation: Taxonomy," *Systems and Control Encyclopedia,* Singh, M.G., ed., Elmsford, NY: Pergamon Press, pp. 4411–4414.

[6] Fishwick, P.A. 1995. *Simulation Model Design and Execution: Building Digital Worlds.* Englewood Cliffs, NJ: Prentice Hall.

[7] Cellier, F.E. 1979. *Combined Continuous System Simulation by Use of Digital Computers: Techniques and Tools,* Doctoral Dissertation, Swiss Federal Institute of Technology, Zurich.

[8] Fishwick, P.A. and Zeigler, B.P. 1992. "A multimodel methodology for qualitative model engineering," *ACM Transactions on Modeling and Computer Simulation* vol. 2, no. 1, pp. 52–81.

[9] Fishwick, P.A. 1993. "A simulation environment for multimodeling," *Discrete Event Dynamic Systems: Theory and Applications* vol. 3, pp. 151–171.

[10] Praehofer, H. 1991. "Systems theoretic formalisms for combined discrete-continuous system simulation," *International Journal of General Systems* vol. 19, no. 3, pp. 219–240.

[11] Praehofer, H., Auernig, F., and Reisinger, G. 1993. "An environment for DEVS-based multiformalism simulation in common Lisp/CLOSS," *Discrete Event Dynamic Systems: Theory and Applications* vol. 3, pp. 119–149.

[12] Mattsson, S.E. and Andersson, M. 1993. "Omola–An object-oriented modeling language," *Recent Advances in Computer-Aided Control Systems Engineering, vol. 9, Studies in Automation and Control,* Jamshidi, M. and Herget, C.J., eds., New York: Elsevier Science Publishers, pp. 291–310.

[13] Widman, L.E., Loparo, K.A., and Nielsen, N.R. 1989. *Artificial Intelligence, Simulation and Modeling,* New York: John Wiley & Sons.

[14] Elzas, M.S., Ören, T.I., and Zeigler, B.P. 1989. *Modelling and Simulation Methodology: Knowledge Systems' Paradigms,* Amsterdam: North Holland.

[15] Cellier, F.E. 1991. *Continuous System Modeling,* New York: Springer-Verlag.

[16] Fishwick, P.A. and Modjeski, R.B., eds. 1991. *Knowledge Based Simulation: Methodology and Application,* New York: Springer-Verlag.

[17] Fishwick, P.A. and Luker, P.A., eds. 1991. *Qualitative Simulation Modeling and Analysis,* New York: Springer-Verlag.

[18] Fishwick, P.A. and Zeigler, B.P. 1991. "Qualitative physics: Towards the automation of systems problem solving," *Journal of Theoretical and Experimental Artificial Intelligence* vol. 3, pp. 219–246.

[19] Birtwistle, G.M. 1979. *Discrete Event Modelling on SIMULA,* Macmillan.

[20] Ege, R.K., ed. 1991. *Object-Oriented Simulation 1991,* Simulation Series, vol. 23, no. 3, San Diego: the Society for Computer Simulation International.

[21] Zeigler, B.P. 1990. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems,* New York: Academic Press.

[22] Booch, G. 1991. *Object Oriented Design,* Benjamin Cummings.

[23] Rumbaugh, J, Blaha, M., Premerlani, W., Frederick, E., and Lorenson, W. 1991. *Object-Oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice Hall.

[24] Kosko, B. 1992. *Neural Networks and Fuzzy Systems.* Englewood Cliffs, NJ: Prentice Hall.

[25] Fishwick, P.A. 1989. "Neural network models in simulation: A comparison with traditional modeling approaches," *Proceedings of the 1989 Winter Simulation Conference.* Washington, DC, pp. 702–710.

[26] Klir, G.J. and Folger, T.A. 1988. *Fuzzy Sets, Uncertainty and Information.* Englewood Cliffs, NJ: Prentice Hall.

[27] Fishwick, P.A. 1991. "Extracting rules from fuzzy simulation," *Expert Systems with Applications* vol. 3, no. 3, pp. 317–327

[28] Fishwick, P.A. 1991. "Fuzzy simulation: Specifying and identifying qualitalive models," *International Journal of General Systems* vol. 9, no. 3, pp. 295–316.

[29] Serra, R. and Zanarini, G. 1990. *Complex Systems and Cognitive Processes,* New York: Springer-Verlag.

[30] Weisbuch, G. 1991. *Complex Systems Dynamics,* Reading, MA: Addison-Wesley.

[31] Langton, C.G., Taylor, C., Farmer, J.D., and Rasmussen, S., eds. 1992. *Artificial Life II,* Reading, MA: Addison-Wesley.

[32] Wolfram, S. 1986. *Theory and Applications of Cellular Automata,* Singapore: World Scientific Publishing (includes selected papers from 1983–1986).

[33] Fujimoto, R.M. 1990. "Parallel discrete event simulation," *Communications of the ACM* vol. 33, no. 10, October, pp. 31–53.

[34] Lin, Y.-B. and Fishwick, P.A. 1996. "Asynchronous parallel discrete event simulation," *IEEE Transactions on Systems, Man and Cybernetics* vol. 26, no. 4, July, pp. 397–412.

[35] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. 1988. *Solving Problems on Concurrent Processors: Volume 1, General Techniques and Regular Problems,* Englewood Cliffs, NJ: Prentice Hall.

[36] McDonald, B. 1992. *Distributed Interactive Simulation: Operational Concept,* Technical Report, UCF Institute for Simulation and Training, Orlando, FL, September.

[37] Loper, M. and Seidensticker, S. 1993. *The DIS Vision: A Map To the Future of Distributed Simulation,* Technical Report, Institute for Simulation and Training, Orlando, FL, October.

[38] Barzel, R. 1992. *Physically-Based Modeling for Computer Graphics,* New York: Academic Press.

[39] Badler, N.I., Barsky, B.A., and Zeltzer, D., eds. 1991. *Making Them Move: Mechanics, Control and Animation of Articulated Figures,* San Mateo, CA: Morgan Kaufmann.

[40] Badler, N.I., Phillips, C.B., and Webber, B.L. 1993. *Simulating Humans: Computer Graphics, Animation and Control,* Oxford, UK: Oxford University Press.

[41] Thalmann, D., ed. 1990. *Scientific Visualization and Graphics Simulation,* New York: Springer-Verlag.

[42] Bass, L., Gornostaev, J., and Unger, C., eds. 1993. *Human-Computer Interaction,* New York: Springer-Verlag, Lecture Notes in Computer Science 753.

[43] *Human Factors in Computing Systems,* 1992 *ACM/SIGCHI Proceedings.*

[44] Grechenig, T. and Tscheligi, M., eds. 1993. *Human-Computer Interaction.* Springer-Verlag, Lecture Notes in Computer Science 733.

[45] Nielsen, J. 1990. *Hypertext and Hypermedia,* New York: Academic Press.

[46] Barrett, E. 1989. *The Society of Text: Hypertext, Hypermedia and the Social Construction of Information,* Cambridge, MA: MIT Press

PAUL A. FISHWICK received a BS degree in mathematics from the Pennsylvania State University, an MS degree in applied science from the College of William and Mary, and a PhD degree in computer and information science from the University of Pennsylvania in 1986. He also has six years of industrial/ government production and research experience working at Newport News Shipbuilding and Dry Dock Co. researching CAD/ CAM parts definition and at NASA Langley Research Center studying engineering database models for structural engineering. He is an Associate Professor in the Department of Computer and Information Sciences at the University of Florida. His research interests are in computer simulation modeling and analysis methods for complex systems. He is a Senior Member of the IEEE and the Society for Computer Simulation International. He is also a member of ACM and AAAI. Dr. Fishwick founded the comp.simulation Internet news group (Simulation Digest) in 1987, which now serves over 15,000 subscribers. He was chairman of the IEEE Computer Society Technical Committee on Simulation (TCSIM) for two years (1988–1990), and he is on the editorial boards of several journals, including the ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics, The Transactions of the Society for Computer Simulation International, International Journal of Computer Simulation, and the Journal of Systems Engineering.

A Methodology for Dynamic Model Abstraction

Kangsun Lee and Paul A. Fishwick

Department of Computer and Information Science and Engineering, University of Florida, Building CSE, Gainesville, Florida 32611 E-mail: kslee@cise.ufl.edu

While complex behavior can be generated through simple systems, as in chaotic and nonlinear systems, complex systems are found where a systems study contains multiple physical objects and interactions. Through the use of hierarchy, we are able to simplify and organize the complex system. Every level within the hierarchy may be refined into another level. System abstraction involves simplification through structural system representation as well as through behavioral approximations of executed model structure. There has been little work on creating a unified taxonomy for model abstraction, and a presentation of such a taxonomy is our main purpose. We present the taxonomy and define two major sub-fields of model abstraction, while illustrating both sub-fields through detailed examples. The introduction of this taxonomy provides system and simulation researchers with a way in which to view and manage complex systems.

1. Introduction

Real world dynamic systems involve a large number of variables and interconnections. Abstraction is a technique of suppressing details and dealing instead with the generalized, idealized model of a system. Computational efficiency and representational economy are main reasons of using abstract models in simulation [1–3] and well as in programming languages [4–6].

Although many diverse areas employ abstraction methods, no agreed-upon taxonomy has been developed to categorize and structure them with underlying characterization of a general approach. Our goal is to clarify how abstraction methods relate to each other under a uniform taxonomy. We define system abstraction to be one of two types: *behavioral* or *structural*. Structural abstraction is a process of organizing the system hierarchically using refinement and homomorphism. Refinement is the process of refining a model to more detailed models of the same type (homogeneous refinement) or different types (heterogeneous refinement), while homomorphism is a mapping that preserves the behavior of the lower-level system under the set mappings [7]. Behavioral abstraction focuses only on behavioral equivalence without structural preservation. In most cases, one should explore both of these methods when constructing systems. For instance, when a system is first being designed, one should construct it hierarchically, with simple model types at first, refining them with more complex model types later. Structural abstraction corresponds to this iterative procedure [8–10]. After creating the hierarchy, we may want to isolate abstraction levels, so a level can be

executed apart from the rest of hierarchy with no detailed internal structure. This is where the behavioral approaches are employed. Below the structural abstraction, each component is black-box with no detailed internal structure. Behavioral abstraction is used to represent those black-boxes by approximating the behavior of the system components. By combining structural and behavioral abstraction together, each level of abstraction is independent from the lower abstraction levels, so a level can be executed apart from the rest of the hierarchy. In depth discussions of each abstraction technique follow in the subsequent sections.

Our contribution is the formulation of a taxonomy capturing two types of abstraction, which have generally been overviewed in separate disciplines. Structural abstraction is found mostly in information on *design,* whereas behavioral abstraction is strewn across many fields of computer science and simulation. Through a unification in terminology, we demonstrate that structural and behavioral methods are complementary aspects of system abstraction. Structural abstraction is common in programming language development within computer science as well as in simulation. Behavioral abstraction is common in statistical analysis and automatic control where system abstractions are used in lieu of more complicated model-based transfer functions. Along with our discussion of the taxonomy, we present examples of each approach to complete the discussion.

The paper is organized as follows: we present the new system abstraction taxonomy with specific methods of each category in Section 2. Then we illustrate the abstraction types using two scenarios and show how abstraction methods perform in both linear and nonlinear system abstraction, in Sections 3 and 4. We close with a summary of the taxonomy and its advantages, with future goals to be achieved.

2. Abstraction Taxonomy

A system consists of data and model components. Data refers to values obtained either by observation or arbitrary assignment of values to model components. Model components, which serve as fundamental building blocks for models, take on the data values. Sample model components include state and event [7]. Figure 1 illustrates our abstraction taxonomy. We sub-define structural abstraction of a system into *data abstraction* and *model abstraction* by the definition of system.

- Data Abstraction: abstraction of input, output, time or parameter system values or time-dependent trajectories.

- Model Abstraction: abstraction of dynamical models.

Data abstraction represents a way of compressing time-dependent information and characterizing a "higher level" of data type. Examples of data abstraction types are symbolic value, statistic mean and variance, interval, ratio and fuzzy sets.

Models must be multi-layered so that different abstraction levels of the model respond to different needs of the analyst [2, 11, 12]. By the method of constructing multi-layered model, we further refine model abstraction to *homogeneous* and *heterogeneous* abstraction. In homogeneous-structural abstraction, dynamical systems are abstracted with only one model type. Each model component is modeled with one model type and refined with the same model type. Selection of specific model type is important and depends on the information that one expects to receive from analysis. For example, one would not choose to model low-level physical behavior with a Petri net since a Petri net is an appropriate model type for a particular sort of condition within a system, where there is contention for resources by discretely-defined moving entities. Examples of methods for homogeneous-structural abstraction are conceptual, declarative, functional, constraint and spatial modeling. Detailed discussion on each model type is shown in [7].

In heterogeneous-structural abstraction, different abstraction levels of a system are provided by allowing either homogeneous or heterogeneous model types together under one structure. To incorporate different levels together, we have constructed a multimodeling methodology [13-17], which provides a way of structuring a heterogeneous and homogeneous set of model types together so that each type performs its part, and the behavior is preserved as levels are mapped [3, 18, 19]. Heterogeneous structural abstraction is equivalent to multimodeling in the sense that we abstract a system structurally mapping one level to another, providing multiple level abstractions. While the multimodel approach is sound for well-structured models defined in terms of state space functions and set-theoretic components, selecting system components in each level is dependent on the next-lowest level due to hierarchical structure. This implies that we are unable to run each level *independently*. It is possible to obtain *output* for any abstraction level but, nevertheless, the system model must be executed at the lowest levels of the hierarchy, since there is where we find the actual functional semantics associated with the model. A new definition and methodology are needed to
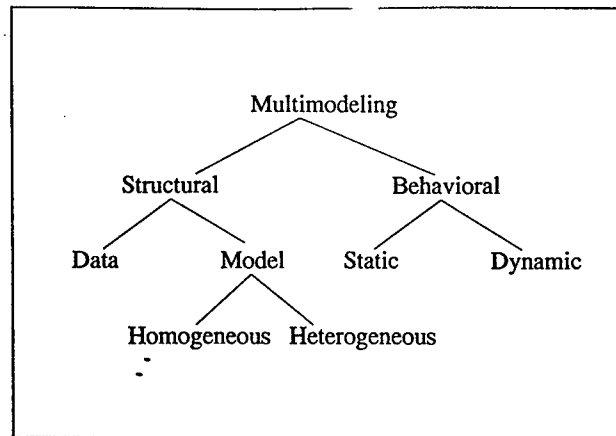
Figure 1. Proposed taxonomy for abstraction

better handle abstraction of systems and components. This is where the behavioral abstraction approaches are employed. By incorporating behavioral abstraction approaches into multimodeling methodology, abstraction in multimodeling allows each level to be understood independently of the others, so that discarding all the abstractions below any given level will still result in a complete behavioral description of a system [8]. This is why we put multimodeling on the top of our abstraction taxonomy.

Behavioral abstraction is where a system is abstracted by its behavior. We replace a system component with something more generic which produces behavior which approximates, to some degree of accuracy, the behavior of the system component at its refined levels. With the help of behavioral abstraction, discarding the refined levels that define a system component will still result in a complete behavioral description of a system [8]. The components are "black boxes" with no detailed internal structure. Behavior is defined as a set of input-output pairs defining a black box. We have two approaches to specifying system behavior:

- Static approach: one takes a system and captures only the steady state output value instead of a complete output trajectory. The input value is defined to be the integral of time value over the simulation trajectory.

- Dynamic approach: one needs to associate time-dependent input and output trajectories.

Though static and dynamic approach describe different allowable behaviors of the same phenomenon, abstraction techniques for dynamic approach can be applied to static approach also. Therefore, we shall focus on dynamical behavioral abstraction for illustrating abstraction techniques.

We denote the output of the dynamical system at time t by $y(t)$ and the input by $u(t)$. The data, defining system behavior, are assumed to be collected in discrete time. At time t we have the data set

$$Z_t = y(1), u(1), \ldots, y(t), u(t) . \tag{1}$$

70

A model of a dynamical system can be seen as a mapping from past data Z^{t-1} to the next output $y(t)$:

$$\hat{y}(t) = \hat{g}(Z^{t-1}). \qquad (2)$$

A "hat" on y is to emphasize that the assigned value is a prediction rather than a measured, "correct" value for $y(t)$. The problem of dynamical behavioral abstraction is to find a mapping $\hat{g}$ that gives good prediction in equation 2 using the information in a data record Z.

System identification [20, 21] is the theory and art of building *mathematical model* of g. Modeling the system consists of selecting a general, parameterized mathematical representation and then tuning the parameters, so that behavior predicted by the model coincides with measurements from the real system. Parameter estimation procedure provides a search through parameter space, effectively, to achieve a close-to optimal mapping between the actual values of the system and the approximate abstract system. Commonly used parameter models are ARX, ARMAX, OE (Output Error) and BJ (Box-Jenkins) [20, 22]. Brief explanations of these models are shown in Sections 3.2.2 and 4.2.1.

Most of the existing identification methods are, in essence, gradient-guided local search techniques which require a smooth search space or a differentiable error energy function. Conventional approaches can thus easily fail in obtaining the global optimum if the multimodel search space is not differentiable or the performance index is not well-behaved in practice [21, 23]. Genetic algorithms represent one way to handle this problem. The genetic algorithm is fine-tuned by simulated annealing, which yields a faster convergence and a more accurate search through the parameter spaces. This global search technique is used to identify the parameters of a system in the presence of

white noise and to approximate a nonlinear multivariable system by a linear time invariant state space model [22].

Neural networks have been established as a general approximation tool for fitting models from input/output data [24–27]. From the system identification perspective, a neural network may be considered as just another model structure [21, 28]. The inputs are linearly combined at the nodes of the hidden layer(s) and then subjected to a threshold-like non-linearity, and then the procedure is repeated until the output nodes are reached. These produce the values that should be matched to the variable $y(t)$ in the observed pair $(y(t), u(t))$. Thus, behavioral abstraction by neural networks is to find $g(t, \theta, u(t))$, where θ represents the weights of the linear combinations involved in network structure. Backpropagation, recurrent and temporal neural networks have been shown to be applicable to system identification [8, 29, 30]. On the other hand, recently introduced *wavelet decomposition* achieves the same quality of approximation with a network of reduced size by replacing the neurons by "wavelons," i.e., computing units obtained by cascading an affine transform and multidimensional wavelets [23].

Table 1 summarizes the base categories along with some sample abstraction techniques discussed so far. Having defined the model abstraction taxonomy, we now proceed to illustrate the different abstraction techniques using the following two scenarios.

3. Sample System I: Boiling Water Example

Consider a pot of water in Figure 2. Here we show a picture of the boiling pot along with an input and output trajectory. The input reflects the state of the knob, which serves to specify external events for the system. The output, T, defines the

Table 1. Sample abstraction categories and associated techniques

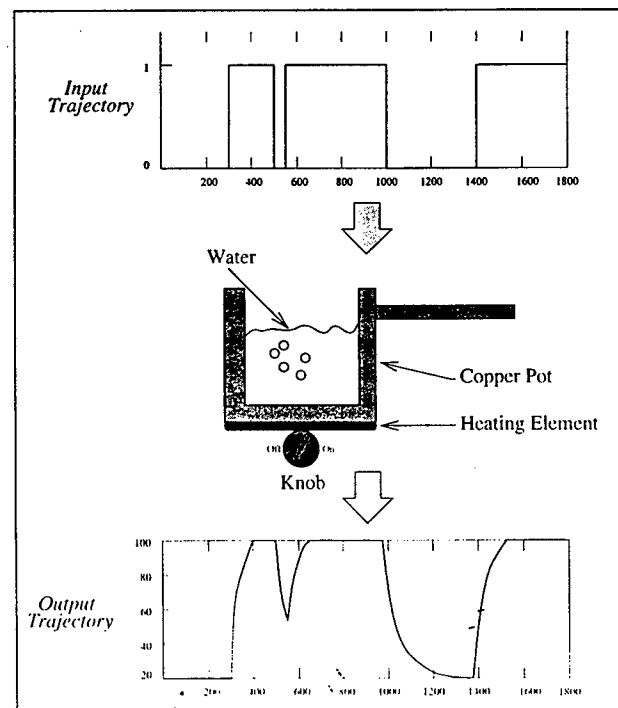| Base Abstraction Type	Abstraction Technique
Data Abstraction	Symbolic Value Mean, Variance Interval, Ratio Fuzzy Number
Structural Abstraction	Conceptual Modeling Declarative Modeling Functional Modeling Constraint Modeling Spatial Modeling
Behavioral Abstraction	Regression System Identification Neural Network Wavelet Genetic Algorithm

Figure 2. Boiling water system

71

temperature of the water over time. Let us define the thermal resistance $R = H/kA$ (H is the height of water, A is the surface area of the pot, and k is the thermal conductivity of water). We will ignore the resistance of the pot since it is not as significant as the resistance of water. The definition for thermal capacitance C is $C\dot{T} = q_{h}$, with q_{h} being the flow of heating element to the water, and C being the total capacitance. Newton's law of cooling states that $Rq_{h} = \Delta T = T_{1} - T_{2}$ where T_{1} is the temperature of the source (heating element), and T_{2} is the temperature of the water. q_{h} is heat flow. Since T_{2} is our state variable, we let $T = T_{2}$ for convenience. By combining Newton's law with the capacitance law, and using the law of capacitors in series, we arrive at

$$k = \frac{C_{1} + C_{2}}{RC_{1}C_{2}} \qquad (3)$$

$$\dot{T} = k(T_{l} - T) . \qquad (4)$$

3.1 Structural Abstraction

The structural approach to system abstraction for the boiling water is defined in a recent text [7] where the boiling water is included as a subsystem within a system of two flasks, and a human operator who mixes the flasks once the liquids are boiling. In the structural abstraction approach to systems, we first need to define our levels of abstraction and then choose which models types to use at each level. In [7, 13, 14], we have the following model levels:

1. Level 1: *Petri net*, defines the action of the human operator and the mixing process.

2. Level 2: *FSA* (*Finite State Automaton*), defines the phases of water during heating and cooling.

    (a) Sub-level 2.1: *FSA*, defines two states: *cold* and *not cold*.

    (b) Sub-level 2.2: *FSA*, defines four states underneath *not cold: heating, cooling, boiling* and *exception*.
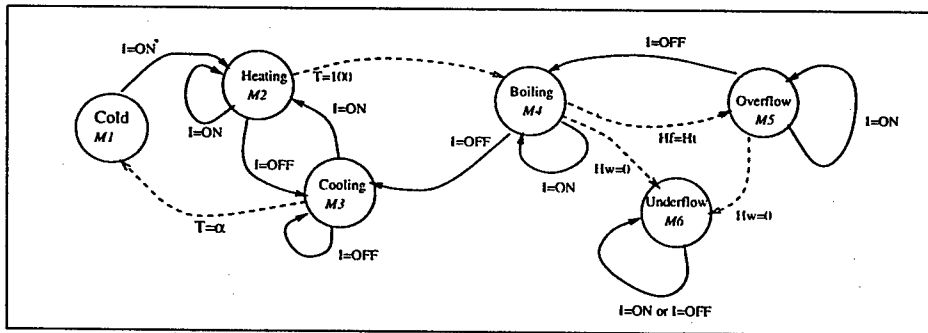
Figure 3. Six state automaton controller for the boiling water multimodel
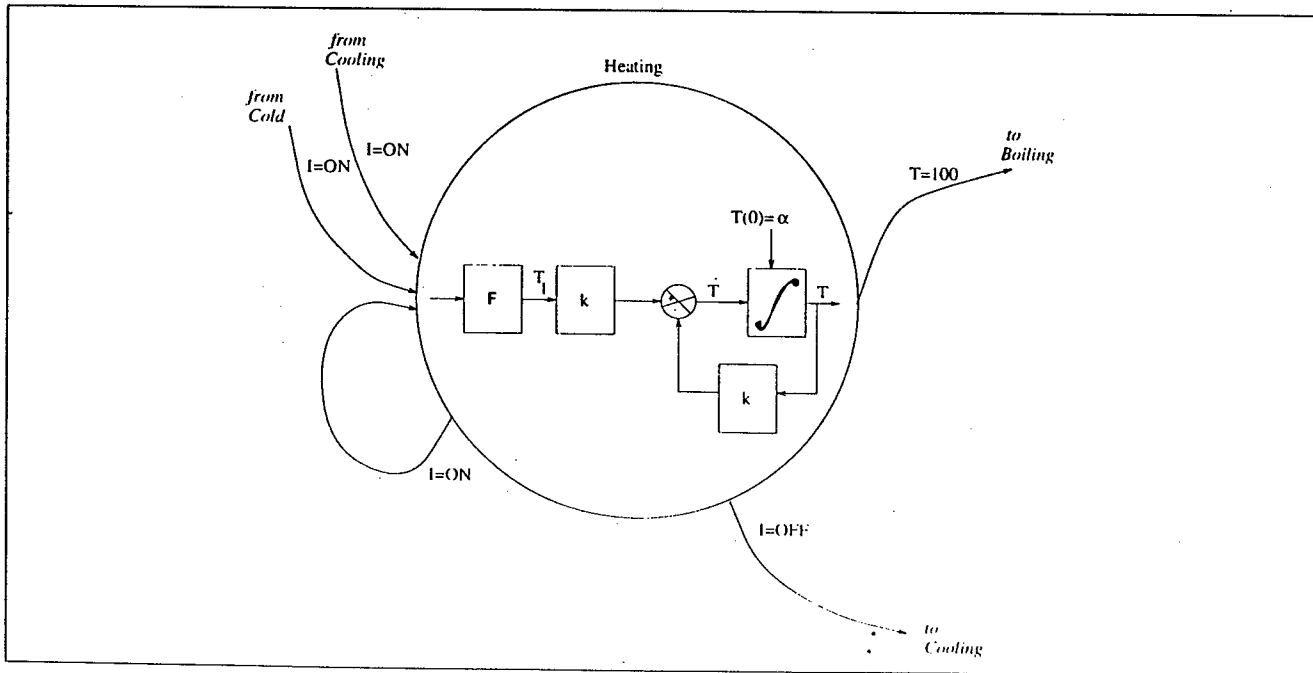
Figure 4. Decomposition of *Heating* state

(c) Sub-level 2.3: *FSA*, defines two states underneath *exception: overflow* and *underflow*.

3. Level 3: *Block model*, defines Newton's Law of Cooling subdefining both *cooling* and *heating* phases.

We show part of the multimodel in Figures 3 and 4. The first model is a compressed version of all the hierarchy specified in Sub-levels 2.1–2.3 above. For the FSA in Figure 3 we choose to represent each state as a continuous model. Specifically, each state will define how three state variables, T (Temperature), H_w (height of water), and H_f (height of foam on the top of the water) are updated. Also the parameter H_t is the height of the pot. I indicates knob's position, and α is the ambient temperature of the water. Figure 4 shows Newton's law of cooling in a functional block form.
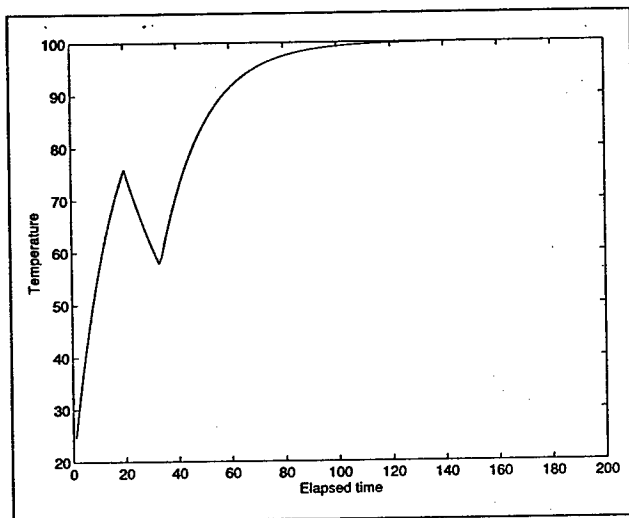
Figure 5. Elapsed time versus temperature
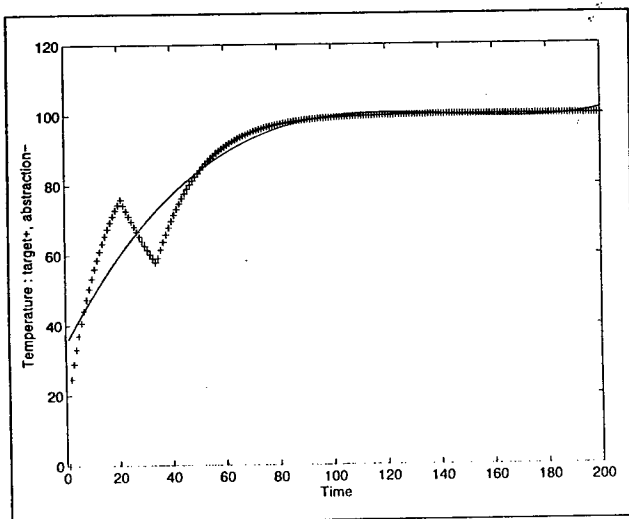
Figure 6. Linear regression

3.2 Behavioral Abstraction

3.2.1 Static Approach. In the static approach, we are interested only in the *final* (i.e., steady state) temperature of the water. A trajectory of final temperature of the water is obtained by varying total amount of simulation time. Figure 5 shows final temperature versus simulation elapsed time. This information is obtained directly from the underlying simulation of the boiling water system. We chose a subset of all possible input time trajectories in such a way that some nonlinearity was introduced into the graph in Figure 5. This was done to challenge the behavioral parameter estimation methods in creating a good fit. This explains why Figure 5 contains a small area of discontinuity in the region between steady state temperature values of 20 and 40. We approximate final temperature by integral value of knob's On/Off trajectory over simulation.

Linear Regression. In general, a polynomial fit to data in vectors x and y(x:input, y:output) is a function, p, of the form

$$p(x) = c_1 x^n + c_2 x^{n-1} + ... + c_d. \qquad (5)$$

The degree is n and the number of coefficients is $d = n + 1$. The coefficients $c_1, c_2, ..., c_n$ are determined by solving a system of simultaneous linear equation: $Ac = y$, where A is matrix whose columns are successive powers of the x vector. Figure 6 shows the result. The approximation is poor in the graph's central region because linear regression is done by polynomial fit, and so it generates a *monotonically increasing* function.

Backpropagation Network. One of the traditional uses of a neural network is *function approximation*. The typical two layer architecture used for a function approximation network is shown in Figure 7. It has one hidden layer of sigmoidal neurons that receive inputs directly and then broadcast their outputs to a layer of linear neurons, which compute the network output [31, 32]. Figure 8 shows the approximation result. This also shows poor performance in abstracting the sharp changing part like the linear regression model.
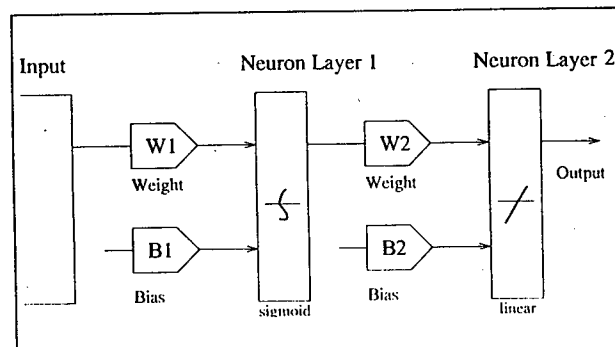
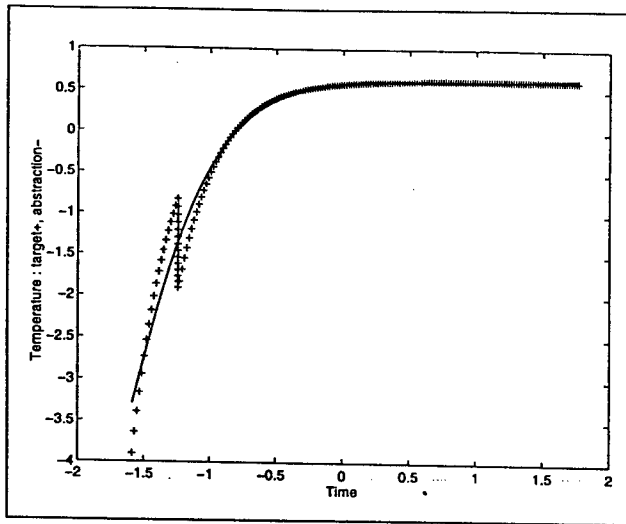Figure 7. Backpropagation network

73

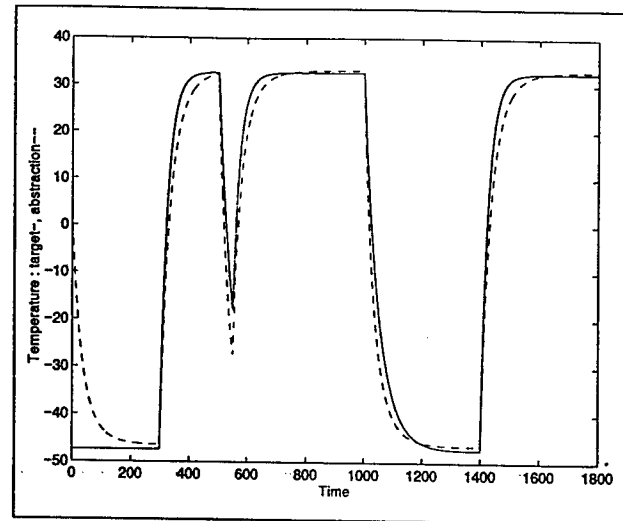**Figure 8.** Backpropagation network for Boiling Water Example

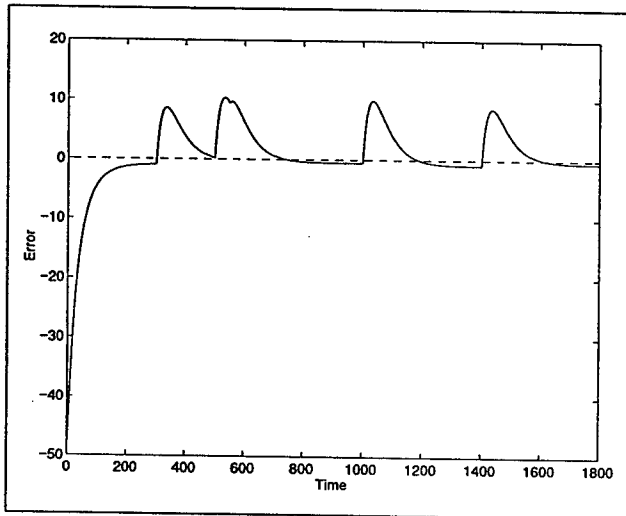**Figure 9.** Box-Jenkins method for Boiling Water Example

Figure 10. Abstraction error in Box-Jenkins method

Figure 11. ADALINE network

3.2.2 *Dynamic Approach.* In the dynamic approach, we are interested in time dependent behavior. In this case, we are concerned not only with the steady state temperature but also the way in which the temperature changes over time. For this approach, we chose a system with just one input and one output, both time-varying trajectories. The input is the input "knob off/knob on" trajectory and the output is the temperature trajectory.

Linear System Identification. The Box-Jenkins method is a frequently used system identification method in time series analysis [20, 26, 27]. Its structure is given by

$$y(t) = \frac{B(q)}{F(q)} u(t - nk) + \frac{C(q)}{D(q)} e(t) \qquad (6)$$

with

$y(t)$: *outputsignal*

$B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb}$

$F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$

$C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$

$D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$

The numbers nb, nc, nd, and nf are the orders of the respective polynomials, and q is the shift operator. The number nk is the

74
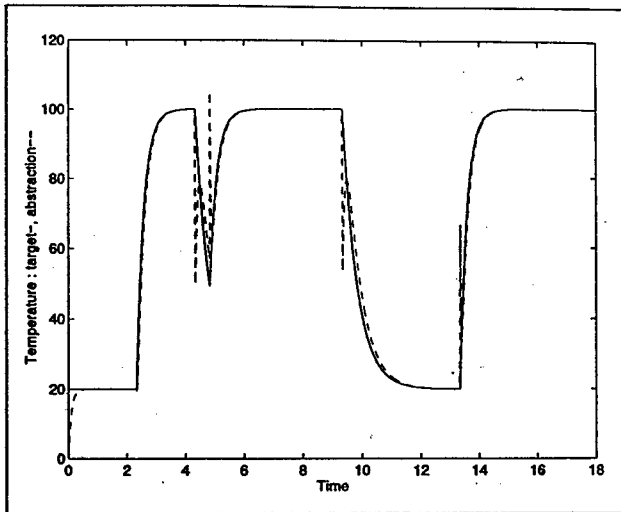
Figure 12. ADALINE network for Boiling Water Example
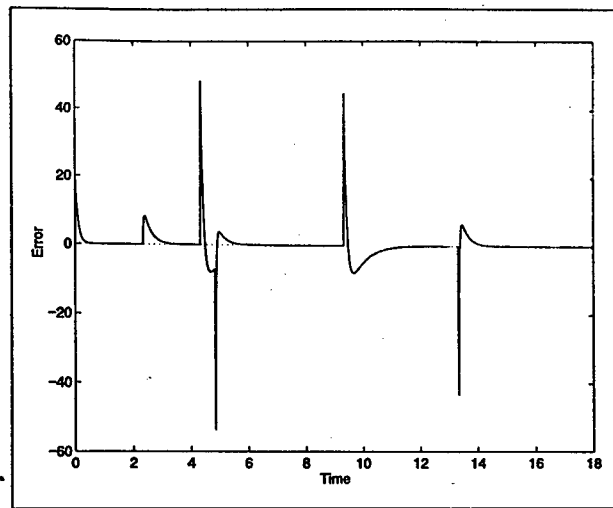
Figure 13. Abstraction error in ADALINE network

number of delays from input to output. Figure 9 shows the approximation result. Successful identification of $y(t)$ depends on how well we guess the values of nb, nc, nd, nf, and nk. Heuristics and "expert rules," if available, aid us in choosing parameters. For example, a large value for a parameter results in computational difficulties to generate $y(t)$, while a small value results in a rough estimation. We often had to tune parameters by hand in order to get a good approximation.

ADALINE Neural Network. ADALINE was developed by Widrow and Hoff [33]. Their neural network model differs from the perceptron [32] in that ADALINE neurons have a linear transfer function. The ADALINE network also enables the Widrow-Hoff learning rule, known as the Least Mean Square (LMS) rule, to adjust weights and biases according to the magnitude of errors. The ADALINE network for our example is shown in Figure 11 with one layer of S neurons connected to R inputs via a matrix of weights W. Figure 12 shows the output signal of the linear neuron with the target signal. An ADALINE neural network takes initial weights and biases, an input signal and a target signal, and then filters the signal adaptively based on input delay and learning rate parameters. In most cases, input delay can be guessed by the modeled system itself. For boiling water example, we know an output at time t is determined by two most recent inputs. However, choosing a good learning rate cannot be implied by the modeled system itself, but through trial and error. Too large a learning rate results in a rough estimation, while too small a value results in severe perturbations during the learning stage.

Gamma Network. The *Gamma* network can be regarded as an extension of the Multilayer Perceptron (MLP). It includes memory structures so that temporal patterns can be converted into static input patterns. A Gamma network focuses on the
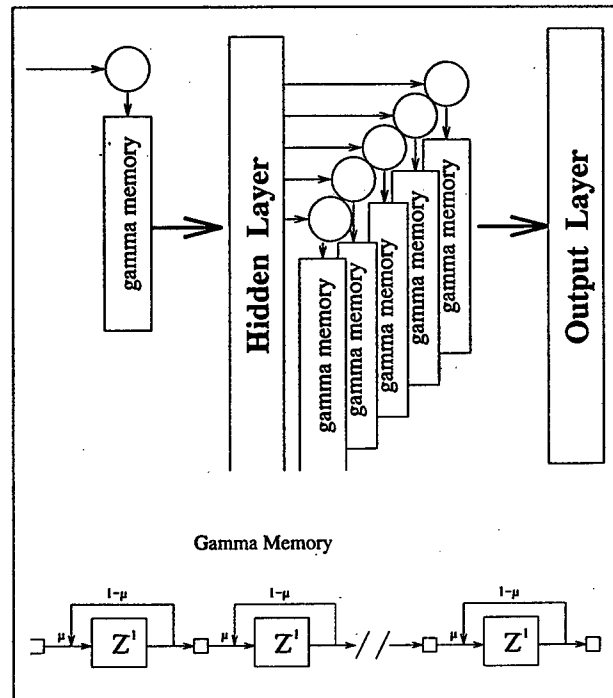
Figure 14. Gamma network

network architecture whose memory structures are implemented only at the input layer to alleviate the difficulty in determining the memory order [34, 35]. A schematic diagram of Gamma network for our example is shown in Figure 14. The abstraction result for the Gamma network is shown in Figure 15. A Gamma network takes a number of hidden nodes, the order of gamma memory, and number of steps for both feed-forwarding and back-propagation, as parameters. Parameter tweaking by hand is needed to accomplish a good abstraction due to the absence of a systematic approach.
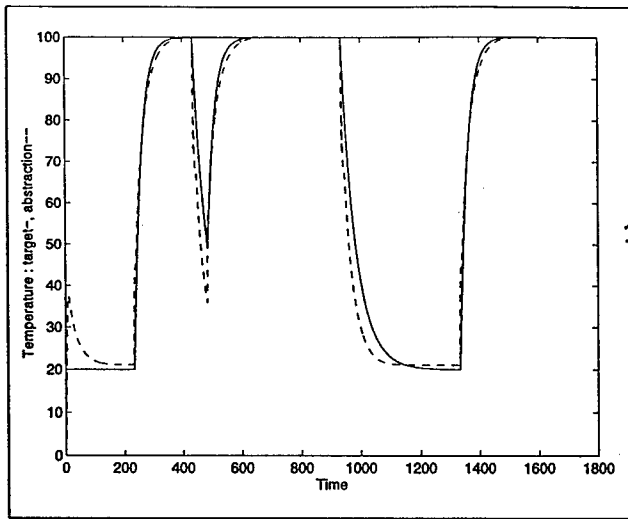
75

Figure 15. Gamma network for Boiling Water Example

Figure 16. Abstraction error in Gamma network

4. Sample System II: Hematopoiesis Model

Though the abstraction methods discussed so far were good at linear system abstraction, non-linear system abstraction is quite different. In this section, we show how these abstraction methods perform under nonlinear conditions.

There are many acute physiological diseases where the initial symptoms are manifested by an alteration or irregularity in a control system which is normally periodic, or by the onset of an oscillation in a hitherto non-oscillatory process. Such physiological diseases have been termed dynamical diseases by Glass and Mackey [36].

Our model deals with the regulation of hematopoiesis, the formation of blood cell elements in the body. Hematopoiesis is the process of blood creation in the body. White and red blood cells are produced in bone marrow. From the marrow they enter the blood circulatory system. As the oxygen level decreases in the body, there is a feedback to the bone marrow—which produces more cells.

4.1 Structural Abstraction

Mackey and Glass [37] provide a delay model for hematopoieses of the following form:

$$\frac{dP(t)}{dt} = \frac{\lambda \theta^n P(t - T)}{\theta^n + P^m(t - T)} - gP(t) \qquad (7)$$

where, λ is the flux of cells into the blood stream, $P(t)$ is the concentration of cells (the population species) in the circulating blood (cells/mm^3), g is day^{-1}, cell loss rate per day, θ is positive constant, and T is maturation delay.

Difference and differential equations are often used in simulation to model low-level phenomena. While the hematopoiesis process can be modeled with the usual differential equation techniques, we need a method for incorporating the delay T between the initialization of cellular production in the bone marrow and the release of mature cells into the bloodstream,
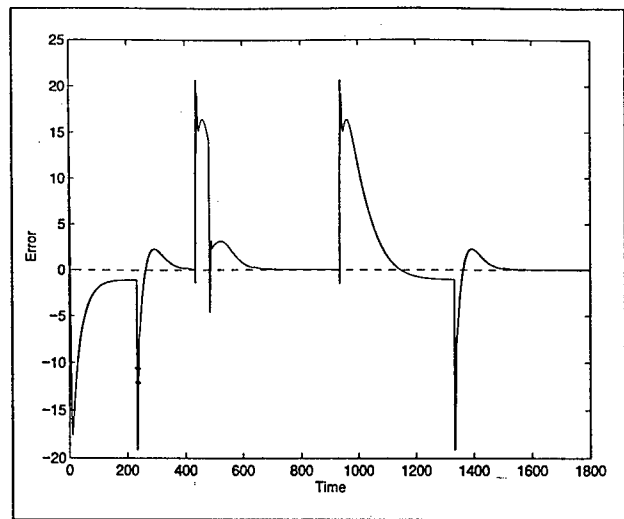
since a state variable's value will stay fixed for a time period. We use λ as an input. Depending on the maturation delay T, we can generate different solutions. Figure 17 displays the time trajectory for the total concentration of blood cells between times 0 to 600 when the delay T = 6 days. As the delay moves upward to T = 20 days, we find a nonperiodic trajectory in Figure 18.

Structural abstraction can be defined by a phase graph depicting $P(t)$ against $P(t - 20)$ as shown in Figure 19. The behavior of the system can be divided into two phases: OUTSIDE_ATTRACTOR and INSIDE_ATTRACTOR. The concept of basins of attraction is a well-formed concept in nonlinear dynamics. In OUTSIDE_ATTRACTOR, the system shows an approximately linear behavior, while, in INSIDE_ATTRACTOR, chaotic behavior appears in the sense that solution pattern is not repetitive in any regular way. Based on this observation, we define structural abstraction of the hematopoiesis model by the following two levels.

- Level 1: *FSA*, defines the chaotic behavior of the system by two phases, OUTSIDE_ATTRACTOR and INSIDE_ATTRACTOR

- Level 2: *Equation model*, subdefines both OUTSIDE_ATTRACTOR and INSIDE _ATTRACTOR phases by equations.

In [7] we describe a method for determining the boundary of an attractor basin for a pendulum with Hamiltonian dynamics. Other methods exist as well for approximating the boundary geometry. For this problem, we will assume that the boundary has been calculated by one of these means. The top level of Figure 20 shows FSA net for Level 1. Transition fires when predicate p becomes true. p is defined to be true when the phase point $p(t)$, $p(t - 20)$ is within the attractor and $\bar{p}$ is otherwise false. We use equation 7 for the subdefinition of both OUTSIDE_ATTRACTOR and INSIDE _ATTRACTOR.
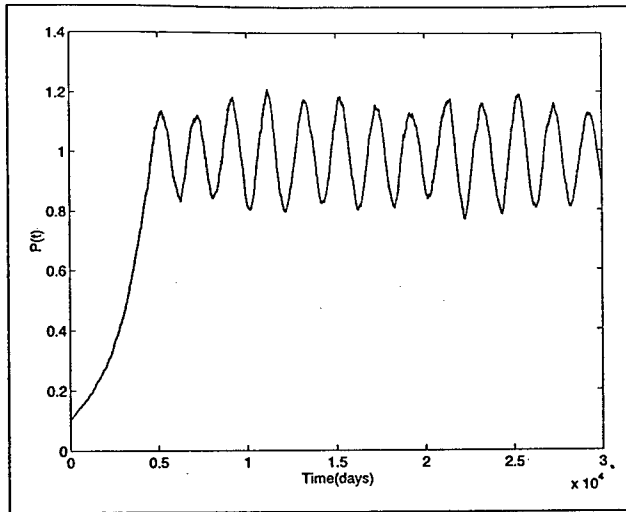
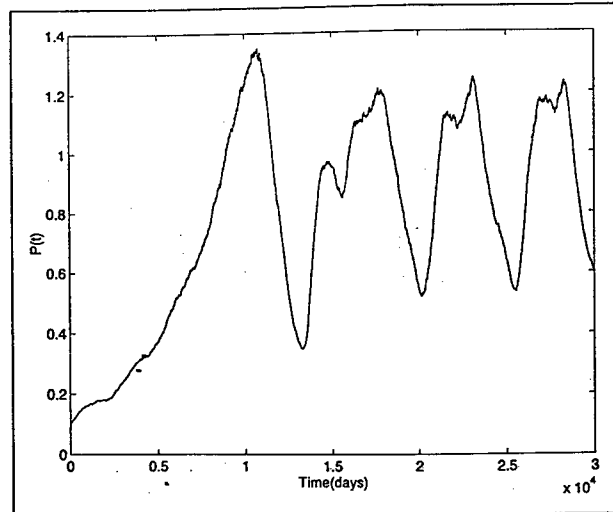**Figure 17.** Cell concentration versus time for delay $T = 6$ days.

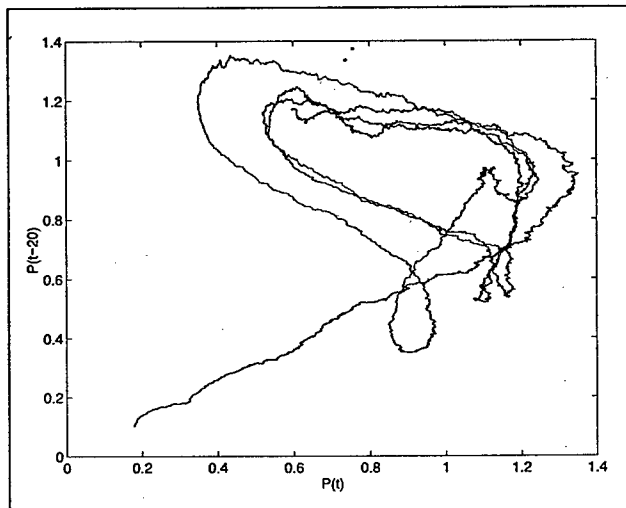**Figure 18.** Cell concentration versus time for delay $T = 20$ days.

Figure 19. Phase graph for delay $T = 20$ days.

Figure 20. Structural abstraction of hematopoiesis model

4.2 *Behavioral Abstraction*

Since we are interested in abstracting the time dependent behavior of cell concentration in the circulating blood, we will restrict out experiments to dynamic-behavioral abstractions. Also, to see how abstraction techniques perform under heavy nonperiodic and nonlinearity conditions, we choose maturation delay 20. Now, the dynamic-behavioral abstraction of hematopoiesis model is to approximate equation 7 with a discrete model of the form

$$P(t) = f(P(t - 1), ..., P(t - na)) \qquad (8)$$

where f is a nonlinear function to be estimated with order na.

Small sampling period for the discretization makes the order of the discrete model very high due to the long dependence of $P(t)$ on $P(t - 20)$, which results in numerical difficulties

to compute the optimal function of f. Therefore, increasing sampling period is needed as long as the discretization is not too rough. Figure 21 displays the time trajectory for the total concentration of blood cells when the sampling period is increased by 100, which introduces more nonlinearity and instability. We choose Figure 21 as the abstraction target and use λ for input.

4.2.1 *System Identification.* A commonly used parametric model is the ARX model:

$$y(t) + a_1 y(t - 1) + ... + a_{na} y(t - na)$$
$$= b_1 u(t - nk) + b_2 u(t - nk - 1) + ... \qquad (9)$$
$$... + b_{nb} u(t - nk - nb + 1) + e(t) .$$

Variables na and nb are the orders of the respective polynomials. The number nk is the number of delays from input to output. We attempted to abstract the hematopoiesis model with a linear ARX model by varying na, nb, and nk, but the results were not satisfactory as shown in Figure 22. Nonlinear model
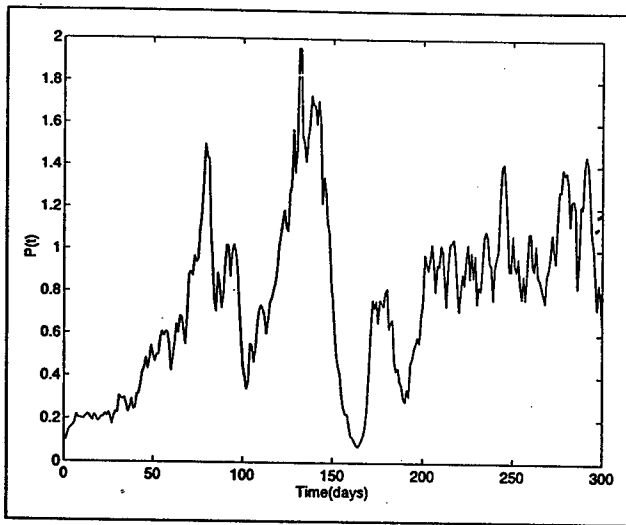
77

Figure 21. Hematopoiesis model for delay = 20 days with increased sampling period: abstraction target
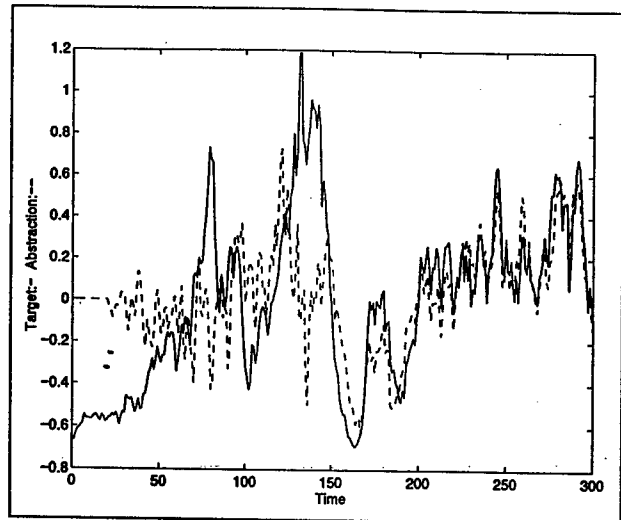

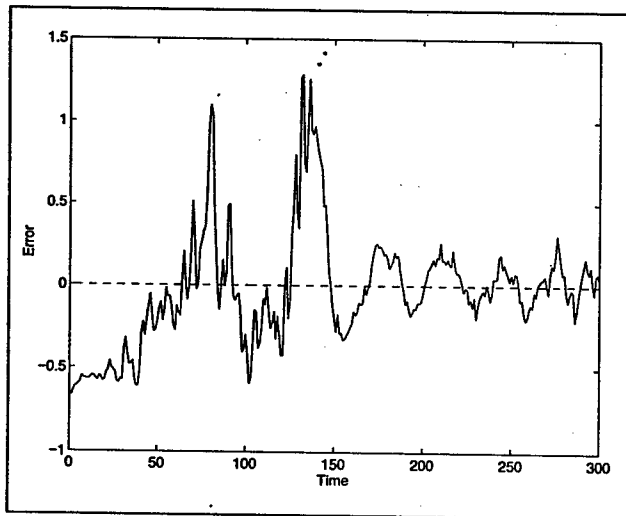**Figure 22. ARX** model for hematopoiesis model
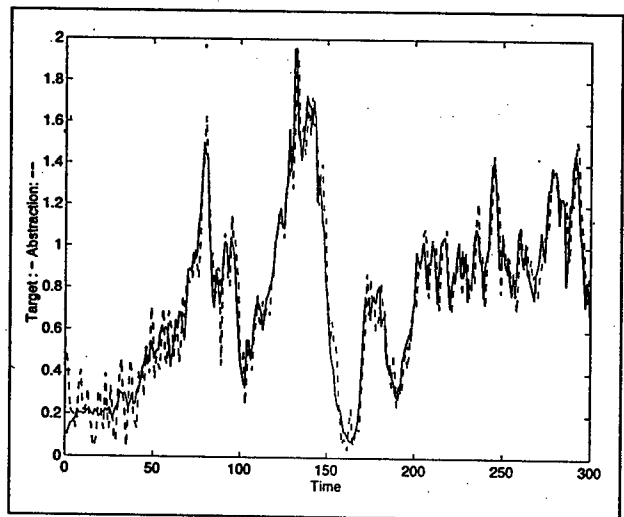

Figure 23. Abstraction error in ARX model

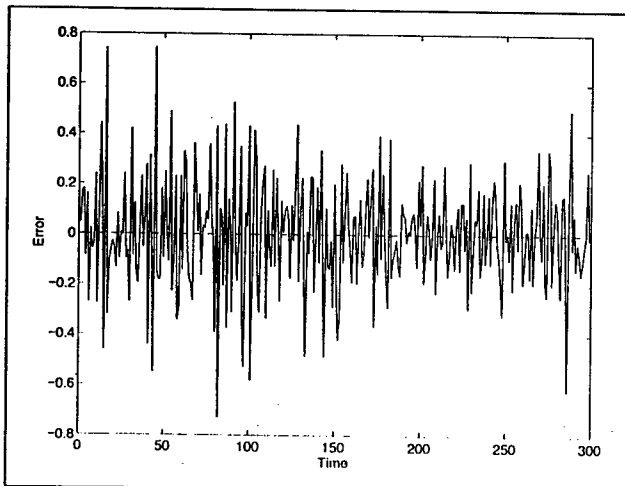**Figure 24. ADALINE** network for hematopoiesis model

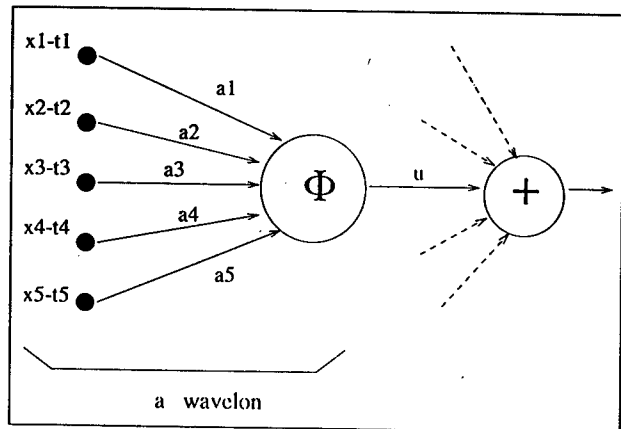**Figure 25.** Abstraction error in ADALINE network

Figure 26. The wavelet network (Dashed arrows figure output connections to other wavelons)

6. Acknowledgments

7. References

[1] Fishwick, P.A. 1987. "A taxonomy for process abstraction in simulation modeling." *IEEE International Conference on Systems, Man and Cybernetics* Alexandria, Virginia, October, vol. 1, pp. 144-151.

[2] Fishwick, P.A. 1989. "Abstraction level traversal in hierarchical modeling." Zeigler, B.P., Elzas, M., and Ören, T. (eds.), *Modelling and Simulation Methodology: Knowledge Systems Paradigms* Elsevier North Holland, pp. 393-429.

[3] Zeigler, B.P. 1972. "Towards a formal theory of modelling and simulation: Structure preserving morphisms." *Journal of the Association for Computing Machinery* vol. 19, no. 4, pp. 742-764.

[4] Berzins, V., Gray, M., and Naumann, D. 1986. "Abstraction-based software development." *Communications of the ACM* vol. 29, no. 5, pp. 402-415.

[5] Booch, G. 1991. *Object Oriented Design With Applications.* The Benjamin/Cummings Publishing Company, Inc.

[6] Sebesta, R.W. 1992. *Concepts of Programming Languages.* The Benjamin/Cummings Publishing Company, Inc.

[7] Fishwick, P.A. 1995. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice-Hall.

[8] Fishwick, P.A. and Lee, K. 1996. "Two methods for exploiting abstraction in systems." *AI, Simulation and Planning in High Autonomous Systems* pp. 257-264.

[9] Fishwick, P.A. 1996. "A taxonomy for simulation modeling based on a computational framework. *IIE Transactions on IE Research.* In Press.

[10] Fishwick, P.A. 1996. "Toward a convergence of systems and software engineering." *IEEE Transactions on Systems, Man and Cybernetics.*

[11] Luh, C.-J. and Zeigler, B.P. 1991. "Abstraction morphisms for task planning and execution." *Proceedings of AI, Simulation and Planning in High Autonomous Systems.* pp. 50-59

[12] Davis, P.K. and Hillestad, R. 1993. "Aggregation, disaggregation, and the challenge of crossing levels of resolution when designing and connecting models." *Proceedings of AI, Simulation and Planning in High Autonomous Systems.* pp. 180-188

[13] Fishwick, P.A. 1991. "Heterogeneous decomposition and coupling for combined modeling." *Winter Simulation Conference,* Phoenix, AZ, December, pp. 1199-1208.

[14] Fishwick, P.A. and Zeigler, B.P. 1992. "A multimodel methodology for qualitative model engineering." *ACM Transactions on Modeling and Computer Simulation* vol. 2, no. 1, pp. 52-81.

[15] Fishwick, P.A. 1992. "An integrated approach to system modelling using a synthesis of artificial intelligence, software engineering and simulation methodologies." *ACM Transactions on Modeling and Computer Simulation* vol. 2, no. 4, pp. 307-330.

[16] Fishwick, P.A. 1993. "A simulation environment for multimodeling." *Discrete Event Dynamic Systems: Theory and Applications* vol. 3, pp. 151-171.

[17] Fishwick, P.A., Narayanan, H., Sticklen, J., and Bonarini, A. 1994. "Multimodel approaches for system reasoning and simulation." *IEEE Transactions on Systems, Man and Cybernetics.*

[18] Fishwick, P.A. 1988. "The role of process abstraction in simulation." *IEEE Transactions on Systems, Man and Cybernetics* vol. 18, no. 1, January/February, pp. 18-39.

[19] Zeigler., B.P. 1990. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* Academic Press.

[20] *System Identification Toolbox.* 1991. The MathWorks, Inc.

[21] Ljung, L. and Soderstrom, T. 1983. *Theory and Practice of Recursive Identification.* Cambridge, MA: MIT Press.

[22] Tan, K.C., Li, Y., Murray-Smith, D.J., and Sharman, K.C. 1995. "System identification and linearisation using genetic algorithms with simulated annealing." *Proceedings of the First IEE/IEEE International Conference on GA in Engineering Systems: Innovations and Applications* pp. 164-169.

[23] Zhang, Q. and Benveniste, A. 1992. "Wavelet networks." *IEEE Transactions on Neural Networks* vol. 3, no. 6.

[24] Cynbenko, G. 1989. "Approximation by superposition of a sigmoidal function." *Mathematics of Control, Signals and Systems* vol. 2, pp. 303-314.

[25] Carrol, S.M. and Dickinson, B.W. 1989. "Construction of neural nets using the radon transform." *IJCNN.*

[26] Tang, Z., de Almeida, C., and Fishwick, P.A. 1991. "Time series forecasting using neural networks vs. Box-Jenkins methodology." *SIMULATION* vol. 57, no. 5, November, pp. 303-310.

[27] Tang, Z. and Fishwick, P.A. 1993. "Feed-forward neural nets as models for time series forecasting." *ORSA Journal of Computing* vol. 5, no. 4, pp. 374-386.

[28] Barron, A.R. 1989. "Statistical properties of artificial neural networks. *Proceedings of the 28th IEEE Conference on Decision and Control.* pp. 280-285.

[29] Rumelhart, D.E., Hinton, G.E., and Williams, R.W. 1986. "Learning internal representations by error propagation." *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* Cambridge, MA: MIT Press.

[30] Mills, P.M., Tade, M.O., and Zomaya, A.Y. 1995. "Identification and control using a hybrid reinforcement learning system." *International Journal in Computer Simulation* vol. 5, pp. 109-126.

[31] *Neural Network Toolbox.* The MathWorks, Inc., 1992.

[32] Fu, L.M. 1994. *Neural Networks in Computer Intelligence.* McGraw-Hill.

[33] Widrow, B. and Sterns, S.D. 1985. *Adaptive Signal Processing.* Prentice-Hall.

[34] de Vries, B. and Principe, J.C. 1992. "The gamma model — A new neural model for temporal processing." *Neural Networks* vol. 5, pp. 565-576.

[35] Principe, J.C. and de Oliveira, P.G. 1993. "The gamma filter — A new class of adaptive IIR filters with restricted feedback." *IEEE Transactions on Signal Processing* vol. 41, no. 2, pp. 649-656.

[36] Murray, J.D. 1990. *Mathematical Biology.* Springer Verlag.

[37] Mackey, M.C. and Glass, L. 1977. "Oscillations and chaos in physiological control systems." *Science* vol. 197, pp. 287-289.

[38] Juditsky, A., Ahang, Q., Delyon, B., Glorennee, P.Y., and Benveniste, A. 1994. *Wavelets in Identification* IRISA publication interne N 849.

[39] Fishwick, P.A. 1996. "Extending object oriented design for physical modeling. *ACM Transactions on Modeling and Computer Simulation.*

KANGSUN LEE received a BS and MS degree in Computer Science from Ewha Womans University, Korea, in 1992 and 1994, respectively. She is currently a PhD student in the Computer and Information Sciences and Engineering Department at the University of Florida, Gainesville. Her research interests are in modeling methodology, abstraction techniques, and simulation. Kangsun Lee's WWW home page is http://www.cise.ufl.edu/~kslee and her E-mail address is kslee@cise.ufl.edu

PAUL A. FISHWICK is an Associate Professor in the Department of Computer and Information Science and Engineering at the University of Florida. He received a PhD in Computer and Information Science from the University of Pennsylvania in 1986. He also has six years of industrial/government production and research experience working at Newport News Shipbuilding and Dry Dock Co. (researching CAD/CAM parts definition) and at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a senior member of IEEE and the Society for Computer Simulation International. He is also a member of the IEEE Society for Systems, Man and Cybernetics, ACM, and AAAI. Dr. Fishwick founded the comp.simulation Internet news group (Simulation Digest) in 1987. He has chaired workshops and conferences in the area of computer simulation, and will serve as General Chair of the 2000 Winter Simulation Conference. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990) and he is on the editorial boards of several journals including the ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics, Transactions of the Society for Computer Simulation, International Journal of Computer Simulation, and the Journal of Systems Engineering. Dr. Fishwick's WWW home page is http://www.cise.ufl.edu/~fishwick and his E-mail address is fishwick@cise.ufl.edu.

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.